# *Session 3:*

# *"Surveys"*

# A Performance Comparison of the Five AES Finalists

Bruce Schneier
Counterpane Internet Security, Inc.
3031 Tisch Way, Suite 100PE
San Jose, CA 95128

Doug Whiting
Hi/fn, Inc.
5973 Avenida Encinas, Suite 110
Carlsbad, CA 92008

15 March 2000

## 1 Introduction

In 1997, NIST announced a program to develop and choose an Advanced Encryption Standard (AES) to replace the aging Data Encryption Standard (DES) [NIS97a,NIST97b]. They solicited algorithms from the cryptographic community, with the intent of choosing a single standard. Fifteen algorithms were submitted to NIST in 1998, and NIST chose five finalists in 1999 [NBD+99]. NIST's plans to choose one (or more than one) algorithm to become the standard in 2000.

NIST's three selection criteria are security, performance, and flexibility. This paper is primarily about the latter two criteria, in light of information regarding the first criterion. In Section 3, we discuss the software performance of the five AES finalists on a variety of CPUs. Detailed information can be found in [SKW+99b], and is not repeated here.

Specifically, we compare the performance of the five AES finalists on a variety of common software platforms: current 32-bit CPUs (both large microprocessors and smaller, smart card and embedded microprocessors) and high-end 64-bit CPUs. Our intent is to show roughly how the algorithms' speeds compare across a variety of CPUs.

There has been considerable discussion in the community about taking the submissions and modifying the number of rounds to increase security. The effect of this is that successful cryptanalytic attacks against a submission do not necessarily knock it out of the running, but instead decrease its performance (by forcing the number of rounds to increase). In [Bih98,Bih99], the author suggests comparing "minimal secure variants" as a way to normalize algorithms.

In Section 4, we give the maximum rounds cryptanalyzed for each of the algorithms, and re-examine all the performance numbers for these variants. We believe that this provides a fairer way of comparing the different algorithms, and the design decisions the different teams made. We then compare the algorithms again, using the minimal secure variants as a way to more fairly align the security of the five algorithms.

## 2 Performance as a Function of Key Length

The speed of MARS, RC6, and Serpent are independent of key length. That is, the time required to set up a key and encrypt a block of text is the same, regardless of whether the key is 128, 192, or 256 bits long. Twofish encrypts and decrypts at a speed independent of key length, but takes longer to set up longer keys.[1] Rijndael both encrypts and decrypts more slowly for longer keys, and takes longer to set up longer keys. The results are summarized in Table 1.

| Algorithm Name | Key Setup | Encryption |
|---|---|---|
| MARS [BCD+98] | constant | constant |
| RC6 [RRS+98] | constant | constant |
| Rijndael [DR98] | increasing | 128: 10 rounds<br>192: 20% slower<br>256: 40% slower |
| Serpent [ABK98] | constant | constant |
| Twofish [SKW+98, SKW+99a] | increasing | constant |

**Table 1: Speed of AES Candidates for Different Key Lengths**

[1] Twofish is more flexible than this chart implies. There are implementations where the encryption speed is different for different key lengths, but they are only suitable for encrypting small text blocks [SKW+98a,WS98,SKW+99a].

In our first performance comparison [SKW+99b], we concentrated on key setup and encryption for 128-bit keys. In this paper, we look at all three key lengths.

# 3 Software Performance

Efficiency on 32-bit CPUs is one of NIST's stated performance criteria. Unfortunately, modern architecture of modern CPUs is so complicated that there is no single performance number that can be used as a basis for comparison.

Today, most high-end microprocessors use 32-bit architectures. These microprocessors range from the Intel Pentium family to embedded CPUs like the ARM family to 32-bit smart card chips. Since all of the AES finalists use 32-bit word sizes, it is not surprising that they are most efficient on these architectures.

The performance space covered by 32-bit CPUs is actually very large, from a 386 or a 68000 or an embedded RISC CPU on the low end, through the various CPUs in the Pentium family. Each CPU chip has its own set of strengths and weaknesses, including clock frequency, cache size and architecture, instruction pipelining (scalar vs. superscalar), etc. Each feature can have significant impact on the speed with which an algorithm can be executed. Even within the same family, major differences in the relative efficiency of certain instruction types can be manifest. As a particular case in point, the Pentium CPU is relatively quite slow at performing multiplies and variable rotates compared to the Pentium Pro/II/III CPUs. This is of interest because two of the AES candidates rely heavily on such operations. It would seem encouraging that the trend is toward better performance on these opcodes in later generations, but in fact future generation processors (such as the IA-64 family) seem to revert back to relatively slow performance for these instructions. The point here is that relative performance of the AES algorithms may vary dramatically on various CPUs. For example, RC6 is the fastest algorithm on the Pentium II/III family by a small margin, but is less half the speed of the fastest candidates on the Pentium and the PA-RISC (and, reportedly, on the Intel Itanium, whose architecture is heavily influenced by PA-RISC). It is not clear exactly what conclusion to draw from such anomalies, particularly since all the candidates are quite fast compared to Triple-DES. In any case, great care should be taken not to assign rankings based solely on a single generation of a single CPU family, which may have a somewhat idiosyncratic performance.

Over the next several years, 64-bit CPUs will become the default microprocessor in desktop computers. These include the Intel Itanium (formerly the Merced) and McKinley, and the DEC Alpha. These microprocessors will first be designed into high-end servers and workstations, and will eventually migrate to commodity desktop computers.

## 3.1 Language Choice

The language of implementation should not matter in theory; i.e., the relative performance of the algorithms in various languages should not depend on the language. In practice, this is at least partially true. For example, some algorithms are very "compiler friendly" and perform quite nicely in high-level languages. To some extent, this is dependent on the language itself, such as the lack of a rotation primitive in C, but to a larger extent it depends on the compiler. The ratio between speeds of Serpent in C and assembler seems to be close to 1:1 on most platforms, while for other algorithms it is between 1.5:1 and 2:1. Thus, Serpent looks relatively much better in C than in assembler. The Java performance seems frankly so dependent on the compiler that the mere phrase "Java performance" seems like an oxymoron. Ultimately, the choice of language may shift the relative performance numbers across algorithms by up to a 2:1 ratio. As long as the metric of choice is somewhat independent to such a range, language really doesn't matter. However, if the ultimate 10% or 50% performance really matters, the algorithm of choice will always be written in assembly—encryption is a discrete and well-defined chunk of code with a single entry point that needs to run fast, a perfect example of something that should be coded in assembly—so those are the best numbers to use in attempting to get precise comparisons.

## 3.2 Comparing Software Performance

The tables in this section compare the five AES finalists on different CPUs. We either used the data in the candidate's AES submission documentation or, where such data was unavailable or unreliable, calculated our own or used the data from other people's implementations [Gla98,Alm99,BGG+99,Gra00,Lip00]. Where we were unsure if a particular optimization technique would work, we gave the algorithm the benefit of the doubt. When there were multiple sources of data, we took the fastest. We believe this is a fair comparison of the algorithms' speeds.

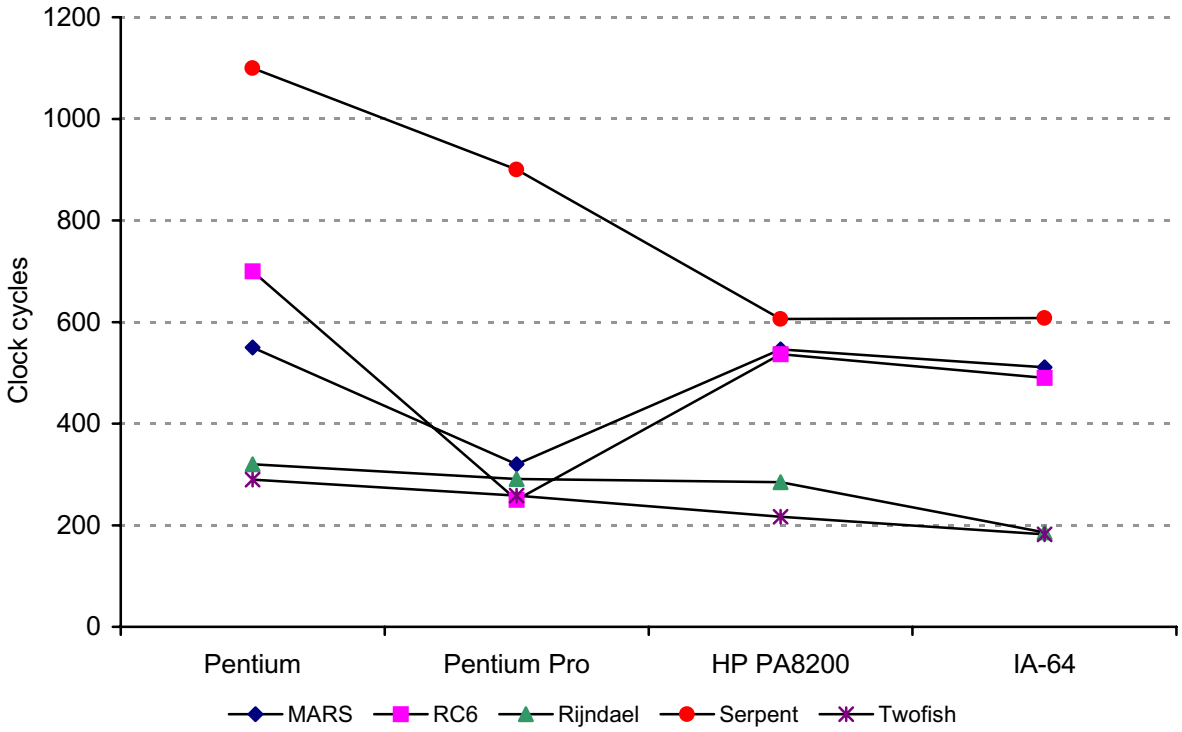



**Figure 1: Encryption Speeds for 128-bit Keys in Assembly**

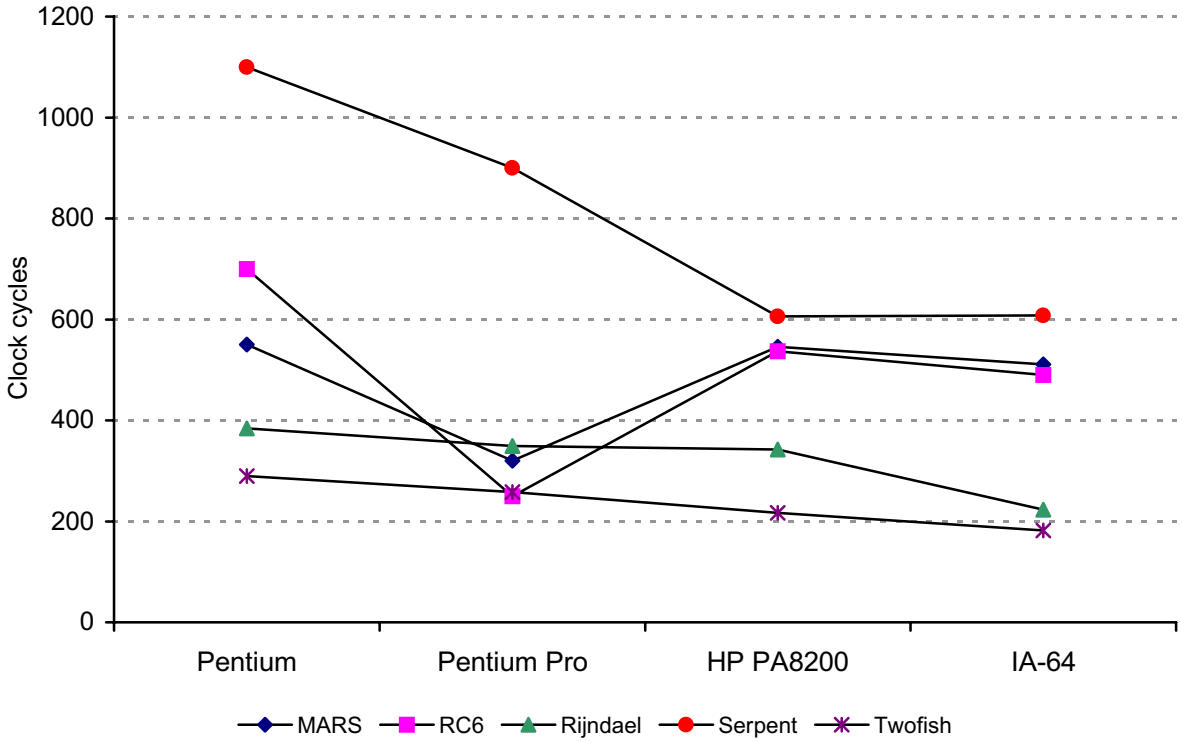



**Figure 2: Encryption Speeds for 192-bit Keys in Assembly**

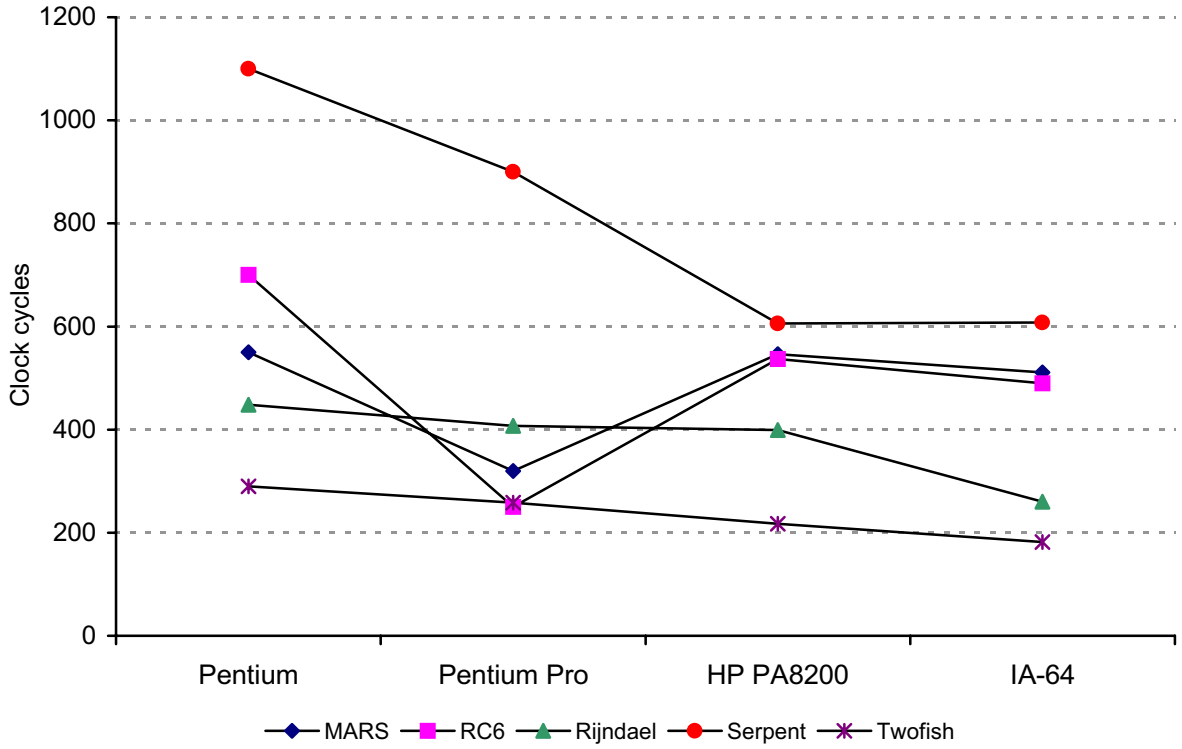



**Figure 3: Encryption Speeds for 256-bit Keys in Assembly**

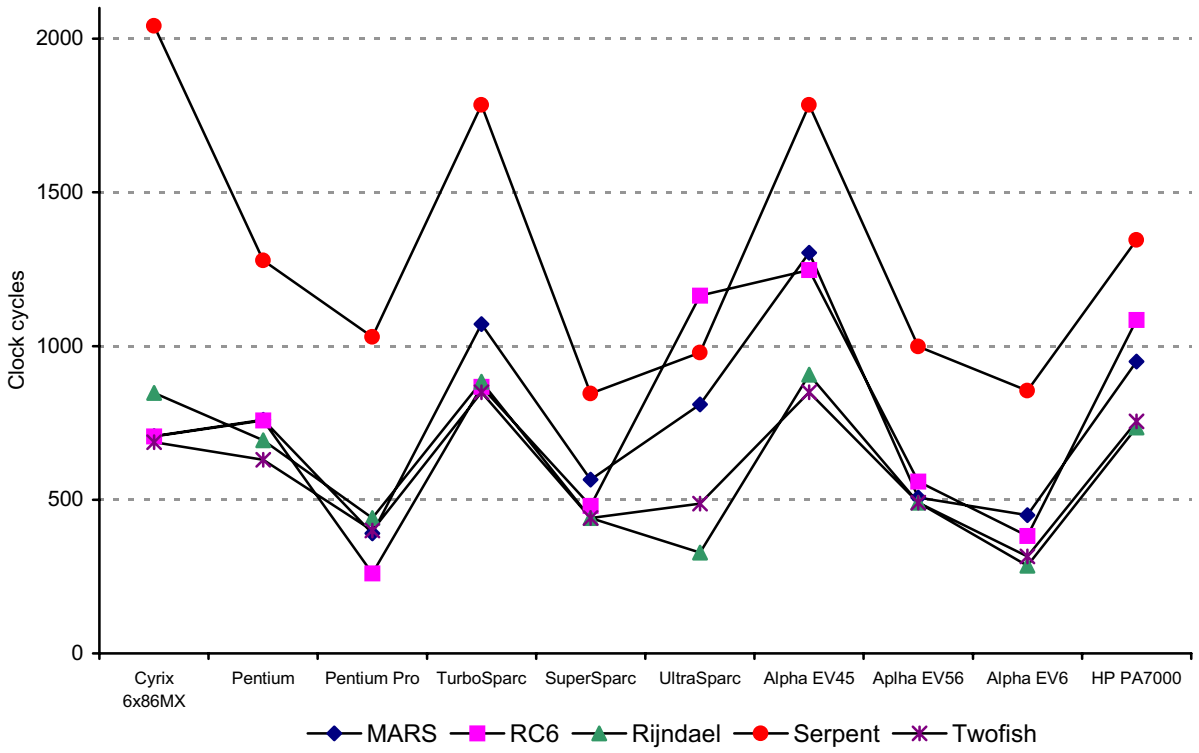



**Figure 4: Encryption Speeds for 128-bit Keys in C**

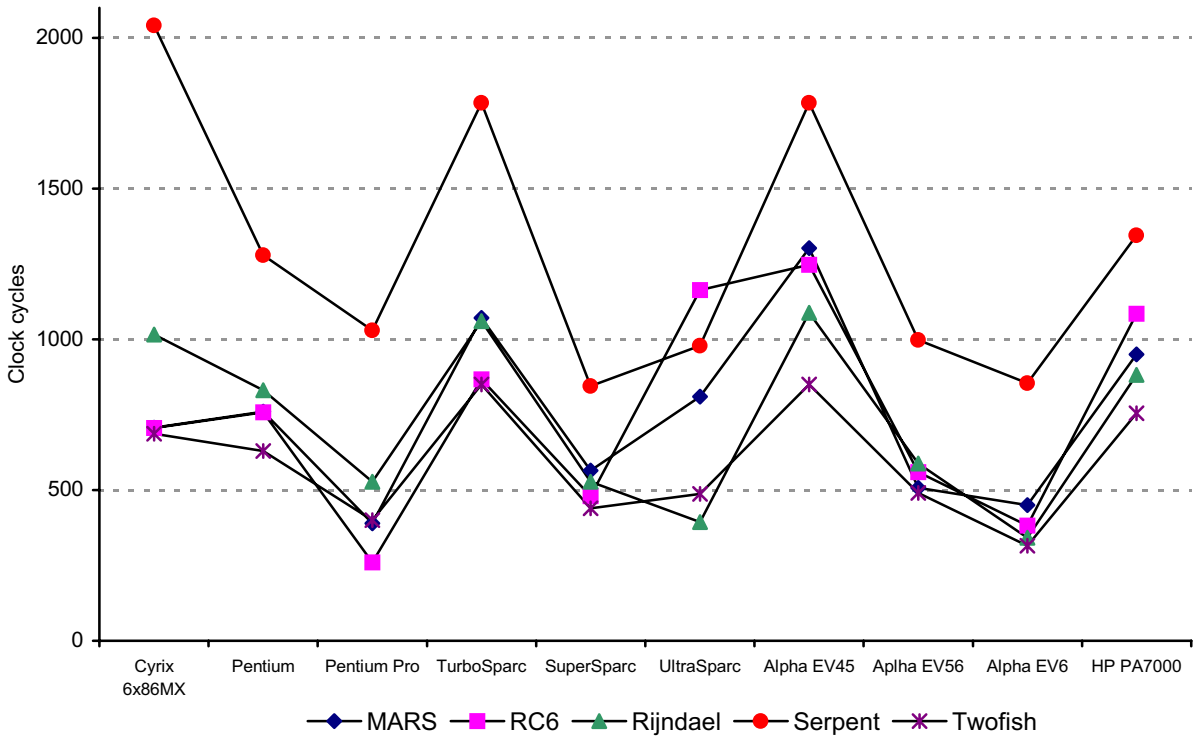



**Figure 5: Encryption Speeds for 192-bit Keys in C**

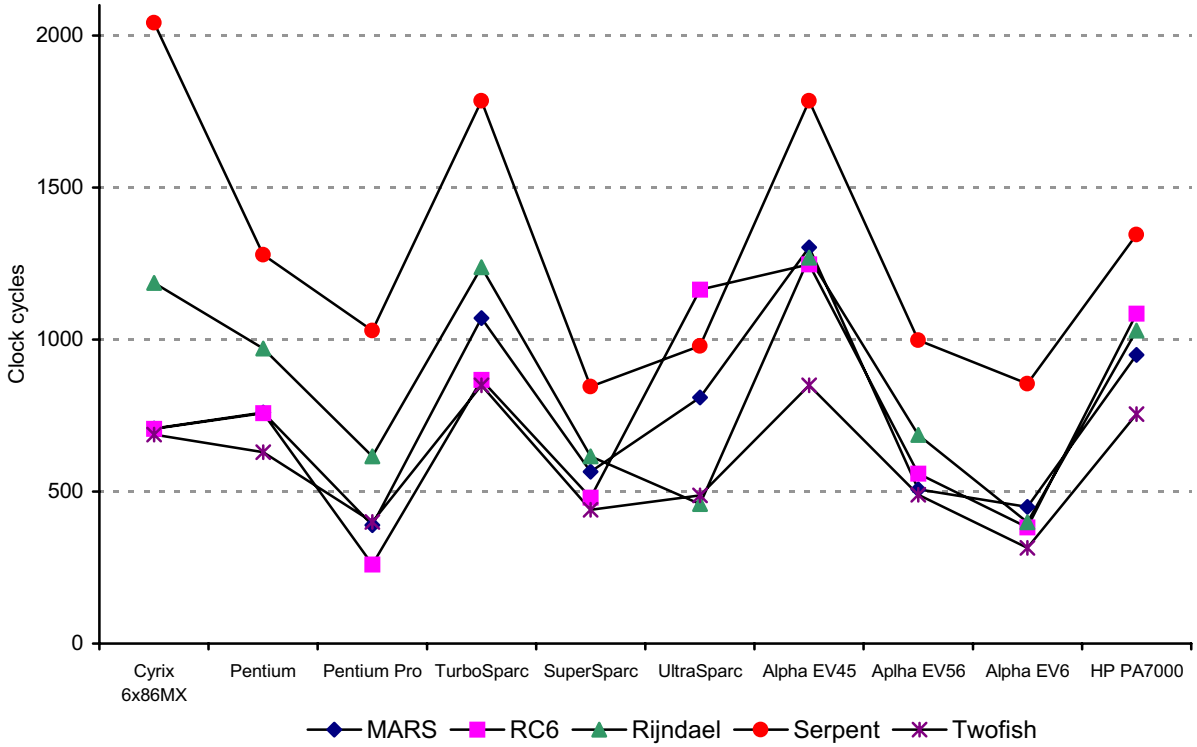



**Figure 6: Encryption Speeds for 256-bit Keys in C**

Interestingly enough, with the exception of MARS and RC6 on Pentium-Pro–class CPUs, the relative performance of the different algorithms was fairly constant. MARS and RC6 are exceptions because they rely heavily on data-dependent rotations and 32-bit multiplications. These operations are not part of the standard RISC instruction-set core, and their performance varies heavily with CPU [SW97]. Normally these two operations are very slow, but on the Pentium Pro, Pentium II, and Pentium III they are fast. Hence, these two algorithms are relatively faster on these CPUs than on both more and less sophisticated CPUs.

For 128-bit keys, Rijndael and Twofish are the fastest algorithms, MARS and RC6 are in the middle, and Serpent is slowest. For larger key lengths, Rijndael gets progressively slower; in some implementations it becomes slower than MARS. These trends hold true in both C and assembly language, although Serpent tends to produce C code that is closer to its assembly-language performance than the others.

### 3.2.1 Key Setup Plus Encryption

For encryption of short blocks of plaintext, performance is a function of both encryption speed and key setup speed. Much less data is available on key setup on various platforms, but we can estimate from the data we have. We use the key setup estimates from [SKW+98b]. Note that the numbers for Twofish take advantage of its flexibility in key setup vs. encryption.

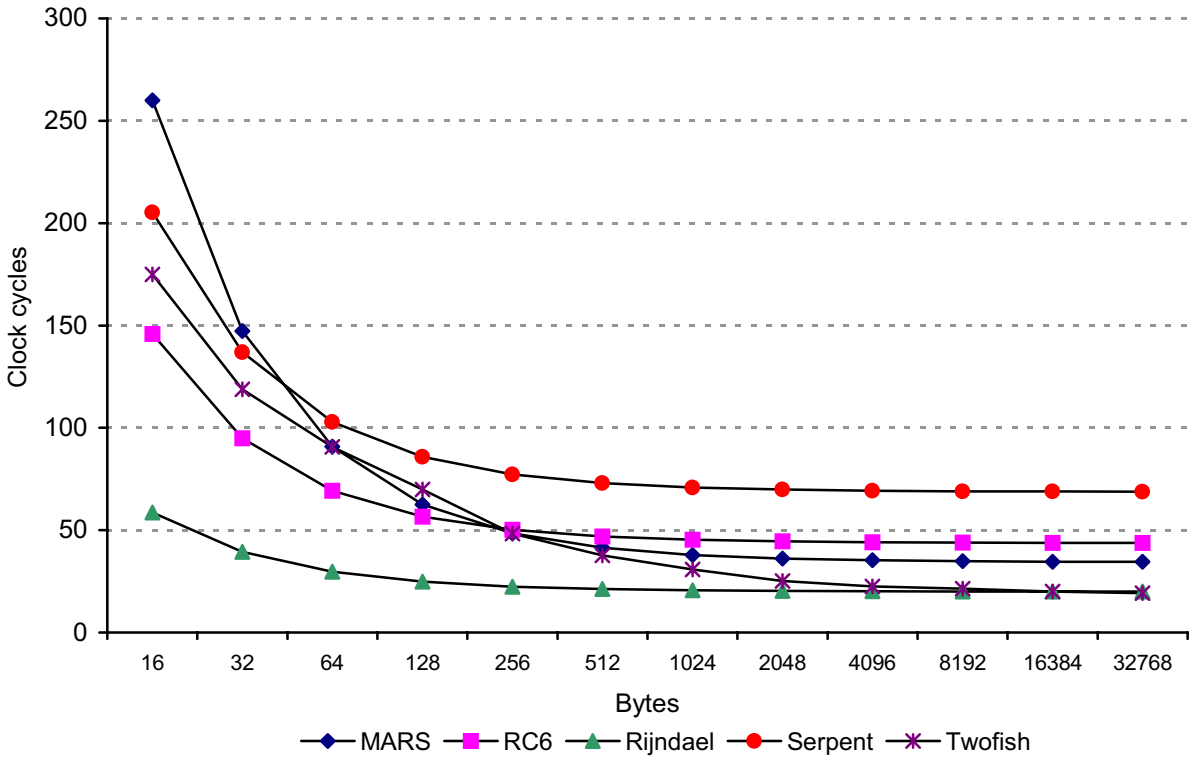



**Figure 7: Key Setup and Encryption Rate, per Byte, for 128-bit Keys on a Pentium in Assembly**

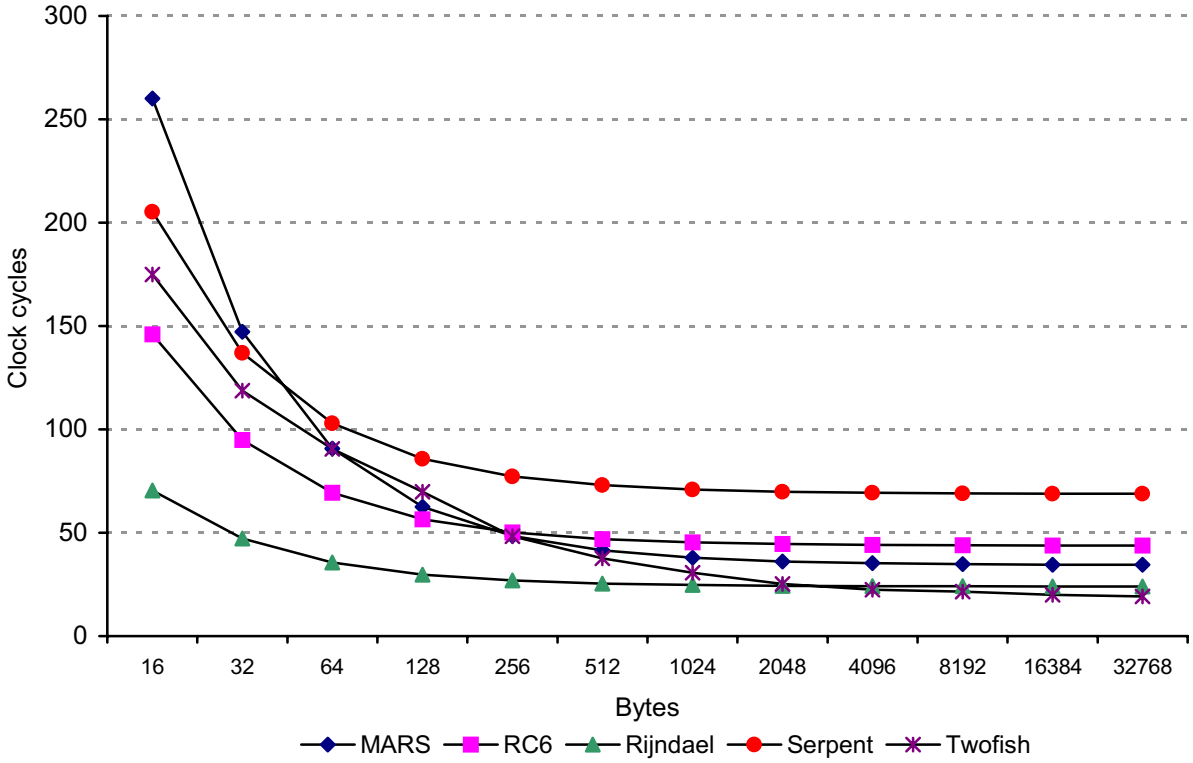



**Figure 8: Key Setup and Encryption Rate, per Byte, for 192-bit Keys on a Pentium in Assembly**

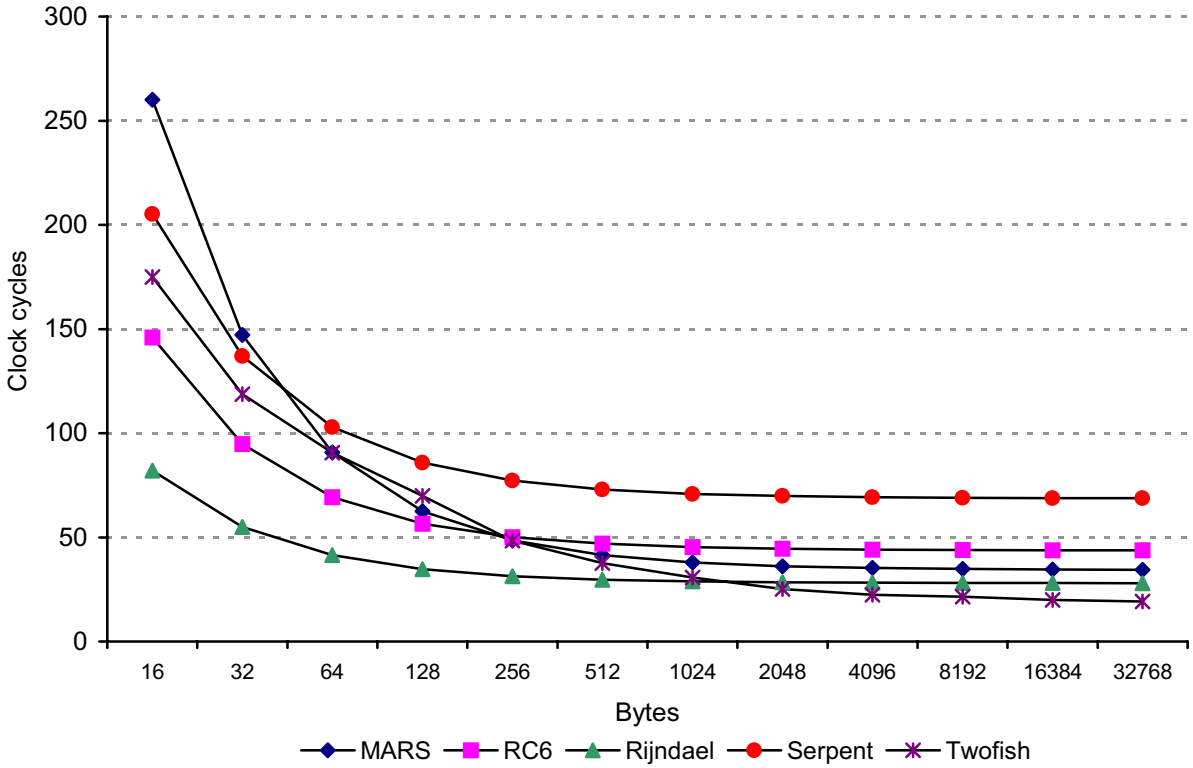



**Figure 9: Key Setup and Encryption Rate, per Byte, for 256-bit Keys on a Pentium in Assembly**

Rijndael is by far the best algorithm for encrypting small blocks, although the algorithms quickly normalize to their "natural" ordering. The 16-byte measure is of particular interest, because that is the speed of the algorithm if used as a hash function.

# 4 Software Performance of "Maximal Insecure Variants"

In [Bih98,Bih99], Biham introduced the notion of comparing the algorithms based on their "minimal secure variants." Different design teams were more or less conservative than each other; the number of rounds in their final submissions was not a fixed percentage of the number of rounds they could successfully cryptanalyze. Biham tried to normalize the algorithms by determining the minimal number of rounds that is secure (either as described by the designers or other cryptographers, or Biham's "best guess"), and then added a standard two cycles.

In his comments to NIST [Knu99], Lars Knudsen presented another rule of thumb for changing the number of rounds of the different algorithms: "Let *r* be the maximum number of rounds for which there is an attack faster than exhaustive key search. Choose *2r* rounds for the cipher." This rule gives us a new, although similar, measure of comparison.

For this comparison, we leave the scaling factor for another discussion, and simply compare the maximal number of rounds for which the best cryptanalytic attack is less complex than a 256-bit brute-force search; call this the "Maximal Insecure Variant." For Table 2, we use the best published cryptanalytic results as explained below.

| Algorithm Name | Rounds |
|---|---|
| MARS | 9 of 16 |
| RC6 | 15 of 20 |
| Rijndael | 8 of 14 |
| Serpent | 9 of 32 |
| Twofish | 6 of 16 |

**Table 2: Maximal Insecure Variants**

If there is a weak key class, we count the attack if the probability of finding the weak key times the complexity of the attack is no more than $2^{256}$. If there is a related-key attack, we make a similar calculation.

- MARS is complicated because there are four different types of round functions. The MARS design team believes that the cryptographic strength of the algorithm is in the "core"; hence, we concentrate on those rounds. There is an 11-round (of 16 total) attack of the MARS core [KKS00a]. There is also an attack against the cipher with the four different round functions symmetrically reduced from 8 rounds to 3 [KS00]. We make the somewhat arbitrary decision to take a scaling "halfway" between those two results: 9 rounds.
- RC6 has an attack against 15 rounds [KM00]. This attack also applies to a weak key class; the attack works for 1 in $2^{60}$ keys, and the complexity of the attack is $2^{170}$. Hence, this attack counts by our definition. The RC6 designers estimate that 16 rounds is attackable, although they give no concrete attack.
- Rijndael has a distinguishing attack against 8 rounds [FKS+00a].
- Serpent has a distinguishing attack against 9 rounds [KKS00a,KKS00b]. The authors estimate that the longest variant that is not as secure as exhaustive search is 15 rounds, although they have no attack.
- Twofish has attacks against 6 rounds. The related-key attacks discussed in [SKW+98,SKW+99a] do not work [FKS+00b].

Unfortunately, these comparisons are fundamentally flawed, because they unfairly benefit algorithms that have been cryptanalyzed the least. Despite almost two years in the public, there has been little cryptanalysis of the five algorithms. We urge caution in reading too much into these numbers, and would like to see further cryptanalysis of the five algorithms.

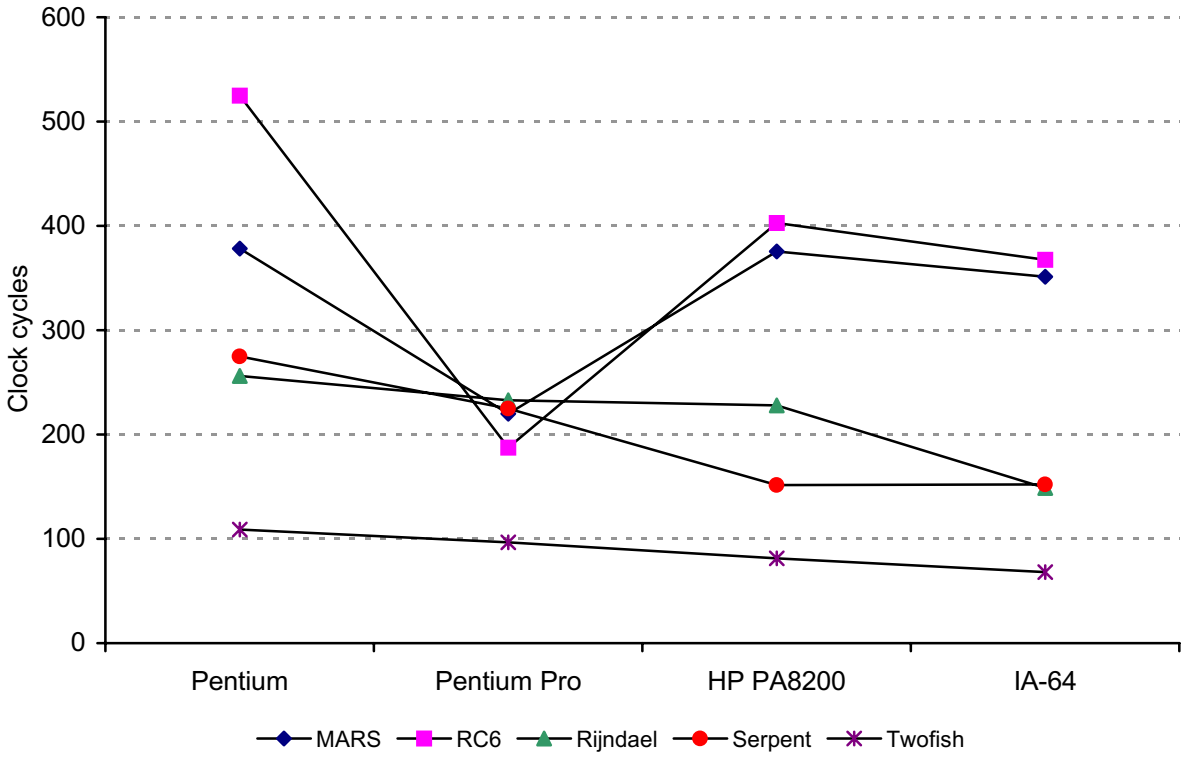



**Figure 10: Encryption Speeds for the Minimal Secure Variant In Assembly**

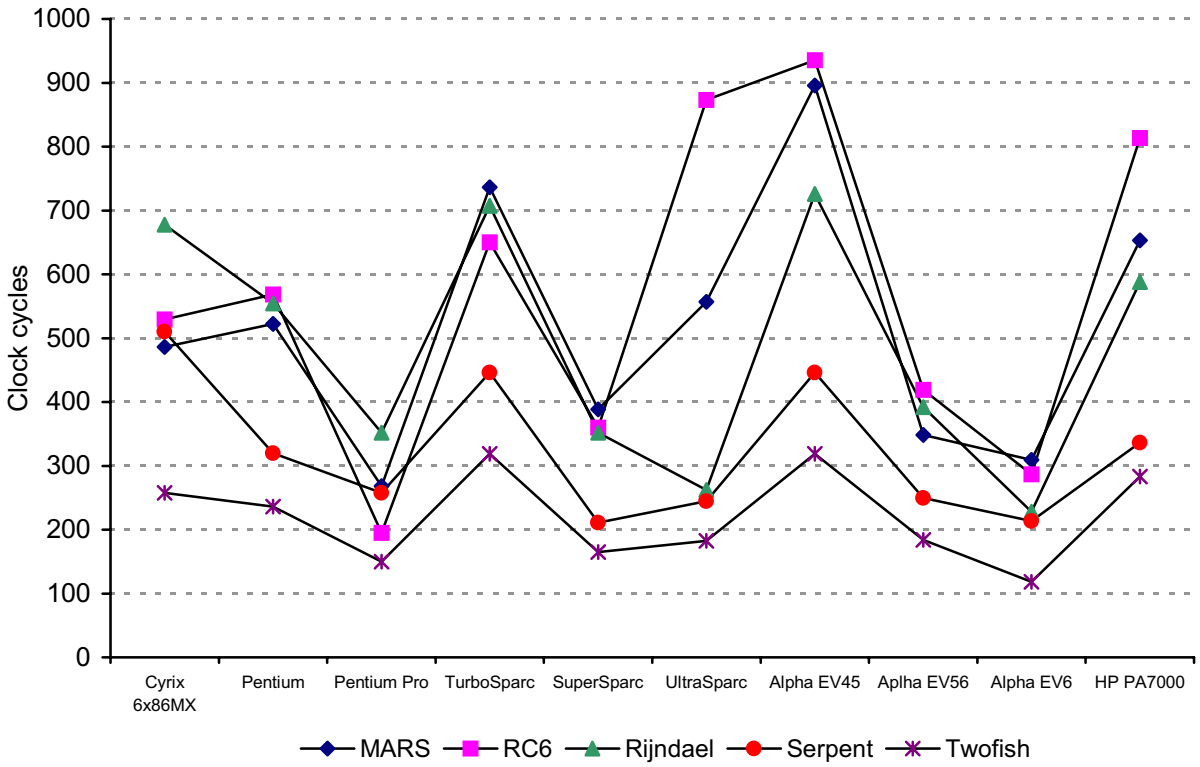



**Figure 11: Encryption Speeds for the Minimal Secure Variant in C**

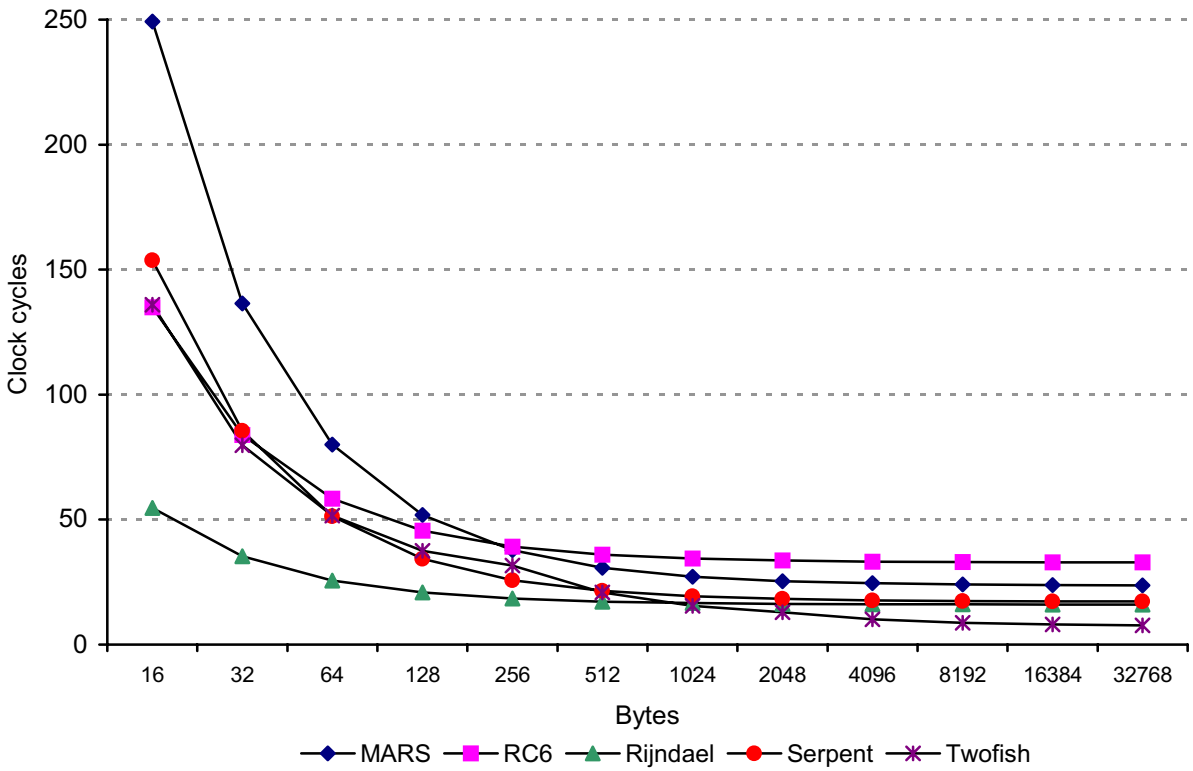



**Figure 12: Key Setup and Encryption Rate, per Byte, for the Minimal Secure Variant on a Pentium in Assembly**

Of course we are not recommending using these round numbers for the algorithms; any AES selection would obviously scale these numbers by a small linear factor. But the relative speeds would be independent of this scaling, and are hence relative to the selection process.

# 5 Performance on Memory-Limited 8-bit Smart Cards

As discussed in [SKW+99b], 8-bit implementations tend to be concerned more with fit than with performance. That is, RAM requirements are more important than clock speed.

Most commodity 8-bit smart card CPUs today include from 128 to 256 bytes of on-board RAM. Each CPU family typically contains members with more RAM and a correspondingly higher cost. RC6 has no effective way to compute subkeys on the fly, thus requiring that all the subkeys be precomputed whenever a key is changed. The RC6 subkeys consume from 176 bytes of "extra" RAM—more than is available on many commonly used CPUs. Although some CPUs include enough RAM to hold the subkeys, it is often unrealistic to assume that such a large fraction of the RAM can be dedicated solely to the encryption function. In a smart-card operating system, for example, encryption is a minor part of the system and won't be able to use more than half of the available RAM. Obviously, if an algorithm does not fit on the desired CPU, with its particular RAM/ROM configuration, its performance on that CPU family is irrelevant.

For some of the candidates, the performance or RAM requirements can depend on whether encryption or decryption is being performed. It is tempting to consider only one of the two operations, using the argument that the smart card can perform the more efficient side of the operation, and the terminal (with the faster, larger, and more RAM-endowed CPU) can perform the less efficient side. Experience shows that this does not work. Many smart card terminals contain a secure module; in many cases this secure module is itself a smart card chip. In several applications, it is a requirement that two smart cards execute a protocol together, and many existing protocols use both encryption and decryption on the same smart card.

Table 3 compares the RAM requirements for the different AES submissions. This table is identical to that in [SKW+99b], with the exception of MARS.

| Algorithm Name | Smart Card RAM (bytes) |
|---|---|
| MARS | 100 |
| RC6 | 210 |
| Rijndael | 52 |
| Serpent | 50 |
| Twofish | 60 |

**Table 3: AES Candidates' Smart Card RAM Requirements**

With its new key schedule "tweak," MARS has on-the-fly subkey generation and appears to require 60 bytes of subkey RAM. When added to the 16 bytes of plaintext and 16 bytes of key, plus other scratch variables, it appears that about 100 bytes of RAM are required. This amount of RAM is much more reasonable than the pre-tweak version, although it is still nearly twice the size of Rijndael, for example.

The new key schedule does allow MARS to fit on small 8-bit CPUs. It should be noted, however, that in this case, the entire rather complex key schedule is recomputed for each block processed, slowing down performance by probably at least a factor of four over a precomputed key schedule

These numbers assume that the key must be stored in RAM, and that the key must still be available after encrypting a single block of text. The results seem to fall into two categories: algorithms that can fit on any smart card (less than 128 bytes of RAM required), and algorithms that can fit on higher-end smart cards (between 128 and 256 bytes of RAM required).[2]

Even if an algorithm fits onto a smart card, it should be noted that the card functionality will include much more than block encryption, and the encryption application will not have the card's entire RAM capacity available to it. So, while an algorithm that requires about 200 bytes of RAM can theoretically fit on a 256-byte smart card, it probably won't be possible to run the smart card application that calls the encryption. For many applications, a RAM requirement of more than 64 bytes just isn't practical.

Given all these considerations, the only algorithms that seem to be suitable for widespread smart card implementation are Rijndael, Serpent, and Twofish.

# 6 Conclusions

The principal goal guiding the design of any encryption algorithm must be security. If an algorithm can be successfully cryptanalyzed, it is not worth using. In the real world, however, performance and implementation cost are always of concern. For most operational systems, encryption is simply another feature that must be incorporated into a design, and must be traded off with other features. Efficiency, whether software encryption speed, hardware gate count, or hardware key-setup speed, may mean the difference between using AES or a home-grown cipher. Therefore, AES must be efficient.

Efficiency means many different things, depending on context. Efficiency may mean bulk encryption speed in software. Encryption may mean key-setup time in hardware. Efficiency may mean hardware gate count, or speed as a hash function, or speed for short messages on 8-bit smart card CPUs. As a standard, AES must be efficient in all of these meanings, since it will be used in all of these applications.

The most obvious conclusion that can be drawn from this exercise is that it is very difficult to compare cipher designs for efficiency, and even more difficult to design ciphers that are efficient across all platforms and all uses. It's far easier to design a cipher to be efficient on one platform, and then let the other platforms come out as they may. In the previous sections, we have tried to summarize the efficiencies of the AES candidates against a variety of metrics. The next thing to do is would be to assign a numerical score to each metric and each algorithm, then weights to each of the metrics, and finally to calculate an overall score for the different algorithms. While appearing objective, this would be more subjective than we want to be; we leave it as an exercise for the reader.

The performance comparisons will most likely leave NIST in a bit of a quandary. The easiest thing for them to do would be to decide that certain platforms are important and others are unimportant, and to choose an AES candidate that is efficient only on the important platforms. Unfortunately, AES will become a standard. This means AES will have to work in a variety of current and future applications, doing all sorts of different encryption tasks. Specifically:

[2] We do not consider the solution of using EEPROM to store the expanded key, as this is not practical in many smart card protocols that require the use of a session key. All algorithms have significantly reduced RAM requirements if this solution is used.

- AES will have to be able to encrypt bulk data quickly on top-end 32-bit and 64-bit CPUs. The algorithm will be used to encrypt streaming video and audio to the desktop in real time.
- AES will have to be able to fit on small 8-bit CPUs in smart cards. To a first approximation, all DES implementations in the world are on small CPUs with very little RAM. They are in burglar alarms, electricity meters, pay-TV devices, and smart cards. Certainly, some of these applications will use 32-bit CPUs as those get cheaper, but that simply means that there will be another set of even smaller 8-bit applications. These CPUs will not go away; they will only become smaller and more pervasive.
- AES will have to be efficient on the smaller, weaker 32-bit CPUs. Smart cards won't be getting Pentium-class CPUs for a long time. The first 32-bit smart cards will have simple CPUs with a simple instruction set.
- AES will have to be efficient in hardware, in not very many gates. There are lots of encryption applications in dedicated hardware: contactless cards for fare payment, for example.
- AES will have to be key agile. There are many applications where small amounts of text are encrypted with each key, and the key changes frequently. IPsec is an excellent example of this kind of application. This is a very different optimization problem than encrypting a lot of data with a single key.
- AES will have to be able to be parallelized. Sometimes you have a lot of gates in hardware, and raw speed is all you care about.
- AES will have to work on DSPs. Sooner or later, your cell phone will have proper encryption built in. So will your digital camera and your digital video recorder.
- AES will need to work as a hash function. There are many applications where DES is used both for encryption and authentication; there just isn't enough room for a second cryptographic primitive. AES will have to serve these same two roles.

Choosing a single algorithm for all these applications is not easy, but that's what we have to do. And when AES becomes a standard, customers will want their encryption products to be "buzzword compliant." They'll demand it in hardware, in desktop computer software, on smart cards, in electronic-commerce terminals, and in other places we never thought it would be used. Anything chosen as AES has to work in all those applications.

# 7 Authors' Biases

As authors of the Twofish algorithm, we cannot claim to be unbiased commentators on the AES submissions. However, we have tried to be evenhanded and fair. Many of the performance numbers in this paper are estimates; we simply did not have time to code each submission in optimized assembly language. As more accurate performance numbers appeared, we have updated the tables in this paper. We will continue to do so.

# References


ABK98 R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 98.

Alm99 K. Almquist, "AES Candidate Performance on the Alpha 21164 Processor," version 2, posted to sci.crypt, 4 Jan 1999.

BGG+99 O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Puopard, J. Stern, and S. Vaudenay, "Report on the AES Candidates," *Second AES Candidate Conference*, 1999.

Bih98 E. Biham, "Design Tradeoffs of the AES Candidates," invited talk presented at ASIACRYPT '98, Beijing, 1998.

Bih99 E. Biham, "A Note Comparing the AES Candidates," revised version, comment submitted to NIST, 1999.

BCD+98 C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS — A Candidate Cipher for AES," NIST AES Proposal, Jun 98.

DR98 J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 98.

FKS+00a N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved Cryptanalysis of Rijndael," *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

FKS+00b N. Ferguson, J. Kelsey, B. Schneier, D. Whiting, "A Twofish Retreat: Related-Key Attacks Against Reduced-Round Twofish," Twofish Technical Report #6, http://www.counterpane.com/twofish-related.html, Feb 00.

Gla98 B. Gladman, "AES Algorithm Efficiency," http://www.seven77.demon.co.uk/aes.htm, 1 Dec 98.

Gra00 L. Granboulan, "AES: Timings of the Best Known Implementations," http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html, 15 Jan 00.

KM00 L. Knudsen and W. Meier, "Correlations in RC6," *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

Knu99 L. Knudsen, "Some Thoughts on the AES Process," comment submitted to NIST, 15 April 1999.

KS00 J. Kelsey and B. Schneier, "Mars Attacks! Cryptanalyzing Reduced-Round Variants of MARS," ," *Third AES Candidate Conference,* 2000, to appear.

KSS00a J. Kelsey, T. Kohno, and B. Schneier, "Amplified Boomerand Attacks Against Reduced-Round MARS and Serpent," *Fast Software Encryption, 7th International Workshop,* Springer-Verlag, 2000, to appear.

KSS00b T. Kohno, J. Kelsey, and B. Schneier, "Preliminary Cryptanalysis of Reduced-Round Serpent," *Third AES Candidate Conference,* 2000, to appear.

Lip00 H. Lipmaa, "AES Ciphers: Speed," http://home.cyber.ee/helger/aes/table.html, 15 Jan 00.

NBD+99 J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback, "Status Report on the First Round of the Development of the Advanced Encryption Standard," *Journal of Research of the National Institute of Standards and Technology,* v. 104, n. 5, Sept.–Oct. 1999, pp. 435–459.

NIST97a National Institute of Standards and Technology, "Announcing Development of a Federal Information Standard for Advanced Encryption Standard," *Federal Register,* v. 62, n. 1, 2 Jan 1997, pp. 93–94.

NIST97b National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," *Federal Register,* v. 62, n. 117, 12 Sep 1997, pp. 48051–48058.

PRB98 P. Preneel, V. Rijmen, and A. Bosselaers, "Principles and Performance of Cryptographic Algorithms," *Dr. Dobb's Journal*, v. 23, n. 12, 1998, pp. 126–131.

RRS+98 R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 98.

SKW+98 B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, Jun 98.

SKW+99a B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

SKW+99b B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions," *Second AES Candidate Conference*, 1999.

WS98 D. Whiting and B. Schneier, "Improved Twofish Implementations," Twofish Technical Report #3, Counterpane Systems, 2 Dec 98.

# Efficiency Testing of ANSI C Implementations of Round 2 Candidate Algorithms for the Advanced Encryption Standard

**Lawrence E. Bassham III**

**Computer Security Division**
**Information Technology Laboratory**
**National Institute of Standards and Technology**

## 1. Introduction

The evaluation criteria for the Advanced Encryption Standard (AES) Round2 candidate algorithms, as specified in the "Request for Comments" [1], includes computational efficiency, among other criteria. Specifically, the "Call For AES Candidate Algorithms" [2] required both Reference ANSI[1] C code and Optimized ANSI C code, as well as Java™[2] code. Additionally, a "reference" hardware and software platform was specified for testing. NIST performed testing on this reference platform, as well as several others. Candidate algorithms were tested for computational efficiency using the Optimized ANSI C source code provided by the submitters.

This paper describes the testing methodology used in ANSI C efficiency testing, along with observations regarding the resulting measurements. The results of the measurements are included followed by conclusions regarding which algorithms have the most consistent performance across different platforms. Some knowledge regarding compilation and processor architectures is useful in understanding how the data was derived. However, the raw data in the document may be useful without necessarily understanding the derivation.

The testing described in this paper is similar to that done in Round 1. The testing has obviously been restricted to the five Round 2 candidates. Additionally, Timing Tests for the Pentium based platforms has been omitted in favor of Cycle Count testing (see Section 3).

## 2. Scope

Performance measurements were taken on multiple platforms. These measurements were analyzed to determine the general rankings of the candidate algorithms with respect to one another. NIST is not interested in the absolute value of the performance measurement, but in the relative value of one algorithm's speed when compared with the rest. From an efficiency point of view, NIST does not intend to rank one algorithm as "better" because it is relatively faster

[1] ANSI – American National Standards Institute

[2] Certain commercial products are identified in this paper. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that material identified is necessarily the best for the purpose.

than another algorithm by 0.5%. However, if one algorithm was faster than another algorithm by 50%, then that would be considered a significant difference. NIST is interested in finding the consistent "top performers" on the test platforms by analyzing the performance data for the algorithms and observing natural breaks.

## 3. Methodology

In the "Call for AES Candidate Algorithms" [2], NIST cited a specific hardware and software platform as the "NIST Analysis Platform" (referred to in this document as the "reference platform") for testing candidate algorithms. This platform consists of an IBM-compatible PC with an Intel® Pentium® Pro™ Processor, 200 MHz-clock speed, 64MB RAM, running Microsoft® Windows® 95, and the ANSI C compiler in the Borland® C++ Development Suite 5.0. Performance measurements were taken on this platform and a large number of additional hardware and software platform combinations. The platforms tested are detailed in Table 1.

**Table 1: System Platforms (Hardware/Software) and Compilers Used in Efficiency Testing**

| Processor/Hardware | Operating System | Compiler |
|---|---|---|
| 200MHz Pentium Pro Processor, 64MB RAM | Windows95 | Borland C++ 5.01 (cycles) |
| | | Visual C++® 6.0 (cycles) |
| | Linux | GCC 2.8.1 (timing) |
| 450MHz Pentium II Processor, 128 MB RAM | Windows98 4.10.1998 | Borland C++ 5.01 (cycles) |
| | | Visual C 6.0 (cycles) |
| 600MHz Pentium III Processor, 128 MB RAM | Windows98 4.10.1998 | Borland C++ 5.01 (cycles) |
| | | Visual C 6.0 (cycles) |
| Sun™: 300MHz UltraSPARC-II™ w/ 2MB Cache, 128 MB RAM | Solaris™ 2.7 (a 64 bit operating system) | GCC 2.8.1 |
| | | Sun Workshop Compiler C™ 4.2 |
| Sun: 2*360MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM | Solaris 2.7 | GCC 2.8.1 |
| | | Sun Workshop Compiler C 4.2 |
| Silicon Graphics™: 2*300MHz R12000™ w/ 4MB Cache, 512 MB RAM | IRIX64™ 6.5.4 (a 64 bit operating system) | GCC 2.8.1 |
| | | MIPSpro C Compiler 7.30 |

Performance measurements were conducted in two different ways. The first performance test method determines the amount of time required to perform cryptographic operations (e.g., how many bits of data can be encrypted in a second, or how many keys can be setup in a second). This type of test is referred to as a "Timing Test" in this document. The second performance testing method counts the number of clock cycles required to perform cryptographic operations (e.g., how many cycles are consumed in encrypting a block of data, or how many cycles are consumed in setting up a key). This type of test is referred to as a "Cycle Count Test" in this document. The Timing Tests utilized the `clock()` timing mechanism in the ANSI C library to calculate the processor time consumed in the execution of the API call and underlying cryptographic operation under test (i.e., `makeKey()`, `blockEncrypt()`, and `blockDecrypt()`). The time consumed to perform a particular operation was then used to calculate the bits/second or keys/second speed measure. The Cycle Count Tests counted the

actual clock cycles consumed in performing the operation under test (for more information on counting clock cycles see [3]). Because cycle counting utilizes assembly language code in the testing program, interrupts could be turned off during testing[3]. This results in a very accurate measure of the performance of the API calls and the underlying cryptographic operations. Additionally, cycle counting eliminates the variability of the processor speed. The same number of clock cycles are required to perform an operation on a 300 MHz Pentium II processor as on a 450 MHz Pentium II processor; there are simply more clock cycles in a second on a 450 MHz-based system. Cycle counting could only be performed on the Intel processor based systems. This is the only processor used by NIST during Round 2 testing that provides access to a true cycle counting mechanism.

### 3.1 Cycle Counting Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) four values are calculated:

- The number of cycles needed to setup a key for encryption;
- The number of cycles needed to encrypt block(s) of data;
- The number of cycles needed to setup a key for decryption; and,
- The number of cycles needed to decrypt block(s) of data.

These values were measured by placing the CPUID and RDTSC assembly language instructions around the NIST API. These instructions were called twice before the cryptographic operation to "flush" the instruction cache (see [3, §3.1]). Additionally, the CLI and STI instructions were used to disable interrupts before testing and enable after testing. This eliminates extraneous interrupts that would skew results. The test program generates 1000 sets of cycle count information as described above for each key size. The values in each category are then sorted, and the median value is determined. A standard deviation is calculated for each test category.

```
makeKey();
cipherInit();
for (r=0; r<1000; r++) {
      cli;              /* Clear Interrupt Flag  */
      cpuid;            /* Clears instruction cache  */
      rdtsc;            /* Read Time Stamp Counter  */
      save counter;
      blockEncrypt();   /*  Perform operation being timed  */
      cpuid;
      rdtsc;            /* Read Time Stamp Counter  */
      subtract counter;
      save counter
      sti;              /* Set Interrupt Flag  */
      }
```

Finally, the average of all values that fall within three standard deviations of the median is determined. This value is the reported average time to perform the specific operation (encrypt, decrypt, or key setup) for a particular key size. Values in this test program are calculated around

[3] Interrupts occur, for example, when the operating system needs to perform some action unrelated to the process that is running. If an interrupt were to occur during cycle count testing, the time spent performing the operating system activity would be included in the time spent on the cryptographic operation. This would lead to inflated and erroneous values for the cycles necessary to perform the cryptographic operation.

the NIST API calls. Results for the Cycle Counting Program can be found in Section 5.1. Pseudo code for the generation of cycle counting information for the `blockEncrypt()` operation is included in Figure 1.

The Cycle Counting Program was run several times with different lengths of data for encryption and decryption to determine if size had any effect on the `blockEncrypt()` and `blockDecrypt()` speeds.

### 3.2 Timing Program

For each key size required by [2] (128 bits, 192 bits, and 256 bits) four values are calculated:

- The time to setup 10,000 keys for encryption;
- The time to encrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit);
- The time to setup 10,000 keys for decryption; and,
- The time to decrypt 8192 blocks of data (8192 blocks*128 bits/block=1048576 bits=1Mbit).

Analysis of this data was performed in the same way as the cycle count program listed above in Section 3.1 (calculation of standard deviation, median, etc.) Results for the Timing Program can be found in Section 5.2. Pseudo code for the generation of timing information for the `blockEncrypt()` operation is included in Figure 2.

```
makeKey();
cipherInit();
for(r=0; r<1000; r++){
      (Start Timer)
      blockEncrypt(8192 blocks);
      (Stop Timer)
      }
```

**Fig. 2**: Pseudo code for Time Testing for `blockEncrypt()`

### 3.2 Compiler Options

***PC***

On the three PCs used during testing, all algorithms were compiled using the same compiler options. Those options and their effect are:

- Borland:
  - -Oi Expand common intrinsic functions
  - –6 Generate Pentium Pro instructions
  - –v Source level debugging (does not effect speed)
  - –A Use only ANSI keywords
  - –a4 Align on 4 bytes
  - –O2 Generate fastest possible code

- Visual C:
  - /G6 Pentium Pro instructions
  - /Ox Best optimization for speed
- Linux/GCC:
  - -O3 Best optimization for speed

The Borland programs were compiled on the 200 MHz Pentium Pro Reference machine. The Visual C and DJGPP programs were compiled on the 450 MHz Pentium II machine. The Linux operating system was installed on a Jaz drive attached to the 200 MHz Pentium Pro Reference machine. Compilations for GCC under Linux were performed on this machine.

***Sun***

All algorithms were compiled using the same compiler options. Those options and their effect are:

- GCC: -O3 Best optimization for speed
- Workshop: -xO5 Best optimization for speed

The compilations for the Sun systems were performed on the 300 MHz UltraSPARC II system.

***SGI***

All algorithms were compiled using the same compiler option. That option and its result is:

- GCC: -O3 Best optimization for speed
- MIPSpro: -O3 Best optimization for speed

The Twofish algorithm compiles on the SGI using the MIPSpro compiler, but results in a Bus Error and a core dump when the `blockEncrypt()` and `blockDecrypt()` functions are invoked. This appears to be a problem with how the compiler is handling byte alignment in the optimized code.

## 4. Observations

Some of the algorithms use flags to determine which compiler is used. By checking which compiler is used, an algorithm may substitute commands that direct the compiler to insert code to make use of instructions available on the CPU. The most common example of this is the use of the ROTL and ROTR instructions to perform left and right logical rotations, respectively. Using the machine instruction to perform these rotations results in code which is two cycles faster than performing the equivalent sequence of using a pair of shifts and an OR operation. This can provide a performance enhancement on various compilers that other algorithms do not enjoy because they do not perform this type of compiler dependent compilation. The Borland compiler does not make use of the machine instructions of ROTL and ROTR. The Visual C compiler can make use of the machine instructions by using the routines `_rotl()` and `_rotr()` to perform the rotation.

The `blockEncrypt()` and `blockDecrypt()` times improved as the numbers of blocks passed to the algorithm at the same time increased, because the API overhead is averaged over more blocks, and more data is available in the cache. The larger amounts of data are still encrypted and decrypted in ECB mode; however, in operational use, Cipher-Block Chaining (CBC) mode would likely be used. Efficiency testing was not performed in CBC mode because this would add another layer of data processing that has no real impact on the performance of the algorithm, i.e., pre- and post-processing the data before calling the algorithms' internal ciphering routines. In addition, there may be performance characteristics from one algorithm to another, based on whether data is treated as two 64-bit blocks or four 32-bit blocks, but this effect depends on the processor characteristics.

# 5. Results

## 5.1 Cycle Count Tables

The values[4] in Ekey, Dkey, Enc, and Dec are all in clock cycles. These values refer to:

- Ekey - The number of cycles needed to setup a 128-bit key for encryption;
- Dkey - The number of cycles needed to setup a 128-bit key for decryption;
- Enc - The number of cycles per block needed to encrypt *n* blocks of data; and,
- Dec - The number of cycles per block needed to decrypt *n* blocks of data.

Note: the data encrypted and decrypted in the cycle count measurements was random (as opposed to using all zero data blocks).

Cycles – Borland C++ 5.01 – 200 MHz Pentium Pro, 64MB RAM, Windows95

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 6815 | 6814 | 1097 | 1049 | 944 | 921 | 937 | 913 | 938 | 914 | 957 | 933 |
| MARS-192 | 7001 | 7001 | 1094 | 1059 | 947 | 921 | 938 | 913 | 937 | 918 | 956 | 935 |
| MARS-256 | 7222 | 7222 | 1081 | 1058 | 944 | 926 | 938 | 913 | 939 | 914 | 958 | 932 |
| RC6-128 | 5171 | 5170 | 950 | 911 | 630 | 576 | 610 | 556 | 614 | 558 | 629 | 582 |
| RC6-192 | 5254 | 5265 | 950 | 914 | 636 | 578 | 609 | 555 | 614 | 558 | 629 | 582 |
| RC6-256 | 5330 | 5331 | 949 | 914 | 630 | 576 | 610 | 556 | 614 | 558 | 629 | 582 |
| RIJNDAEL-128 | 2208 | 2870 | 826 | 836 | 690 | 690 | 685 | 686 | 682 | 681 | 704 | 714 |
| RIJNDAEL-192 | 2972 | 3786 | 958 | 961 | 823 | 815 | 815 | 808 | 820 | 811 | 850 | 835 |
| RIJNDAEL-256 | 3691 | 4684 | 1106 | 1137 | 982 | 996 | 939 | 946 | 939 | 947 | 961 | 968 |
| SERPENT-128 | 12324 | 12291 | 3569 | 3273 | 3429 | 3158 | 3422 | 3155 | 3422 | 3163 | 3436 | 3178 |
| SERPENT-192 | 14389 | 14398 | 3574 | 3301 | 3429 | 3159 | 3420 | 3147 | 3424 | 3165 | 3438 | 3176 |
| SERPENT-256 | 16639 | 16644 | 3570 | 3214 | 3429 | 3074 | 3420 | 3064 | 3425 | 3163 | 3438 | 3175 |
| TWOFISH-128 | 13544 | 13372 | 1052 | 1009 | 725 | 681 | 706 | 660 | 708 | 662 | 727 | 687 |
| TWOFISH-192 | 15707 | 15544 | 1052 | 993 | 722 | 675 | 706 | 660 | 708 | 663 | 728 | 686 |
| TWOFISH-256 | 21344 | 21181 | 1049 | 996 | 723 | 679 | 704 | 660 | 708 | 661 | 729 | 682 |

[4] The relative uncertainty for values in all tables is ≤ 1%.

Cycles – Visual C 6.0 – 200 MHz Pentium Pro, 64MB RAM, Windows95

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 4964 | 4964 | 837 | 754 | 687 | 598 | 681 | 593 | 684 | 595 | 718 | 629 |
| MARS-192 | 4996 | 4996 | 821 | 737 | 686 | 601 | 680 | 593 | 683 | 596 | 719 | 629 |
| MARS-256 | 5185 | 5185 | 823 | 743 | 689 | 601 | 680 | 593 | 682 | 595 | 720 | 629 |
| RC6-128 | 2293 | 2294 | 640 | 627 | 351 | 351 | 340 | 332 | 343 | 334 | 382 | 355 |
| RC6-192 | 2401 | 2402 | 640 | 627 | 352 | 351 | 340 | 332 | 343 | 334 | 382 | 355 |
| RC6-256 | 2512 | 2513 | 642 | 629 | 352 | 351 | 343 | 332 | 343 | 334 | 382 | 355 |
| RIJNDAEL-128 | 1278 | 1764 | 1277 | 1308 | 1138 | 1133 | 1125 | 1136 | 1134 | 1135 | 1149 | 1124 |
| RIJNDAEL-192 | 2002 | 2566 | 1512 | 1574 | 1368 | 1362 | 1358 | 1365 | 1361 | 1372 | 1388 | 1365 |
| RIJNDAEL-256 | 2591 | 3257 | 1732 | 1798 | 1604 | 1596 | 1591 | 1599 | 1596 | 1601 | 1614 | 1588 |
| SERPENT-128 | 7092 | 7104 | 1439 | 1293 | 1298 | 1135 | 1286 | 1129 | 1285 | 1128 | 1326 | 1165 |
| SERPENT-192 | 9048 | 9035 | 1455 | 1294 | 1295 | 1135 | 1285 | 1126 | 1285 | 1126 | 1326 | 1168 |
| SERPENT-256 | 10861 | 10850 | 1454 | 1275 | 1292 | 1135 | 1285 | 1127 | 1286 | 1128 | 1326 | 1166 |
| TWOFISH-128 | 9950 | 9790 | 1264 | 1024 | 965 | 725 | 947 | 707 | 950 | 711 | 967 | 740 |
| TWOFISH-192 | 13298 | 13136 | 1265 | 1020 | 966 | 728 | 947 | 707 | 949 | 721 | 965 | 753 |
| TWOFISH-256 | 18555 | 18394 | 1278 | 1016 | 965 | 726 | 947 | 707 | 950 | 710 | 966 | 743 |

Cycles – Borland C++ 5.01 – 450 MHz Pentium II, 128MB RAM, Windows98

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 6837 | 6837 | 1105 | 1082 | 947 | 924 | 939 | 913 | 941 | 920 | 986 | 963 |
| MARS-192 | 7040 | 7038 | 1105 | 1092 | 949 | 919 | 939 | 913 | 937 | 921 | 985 | 961 |
| MARS-256 | 7249 | 7249 | 1105 | 1082 | 949 | 922 | 936 | 914 | 941 | 921 | 992 | 966 |
| RC6-128 | 5186 | 5183 | 984 | 944 | 631 | 578 | 610 | 556 | 617 | 560 | 651 | 598 |
| RC6-192 | 5279 | 5279 | 984 | 943 | 631 | 577 | 609 | 555 | 617 | 560 | 651 | 598 |
| RC6-256 | 5363 | 5364 | 984 | 944 | 631 | 578 | 609 | 555 | 617 | 560 | 651 | 598 |
| RIJNDAEL-128 | 2254 | 2912 | 845 | 844 | 689 | 699 | 681 | 692 | 696 | 697 | 777 | 783 |
| RIJNDAEL-192 | 2994 | 3778 | 983 | 993 | 818 | 814 | 811 | 807 | 826 | 820 | 892 | 896 |
| RIJNDAEL-256 | 3722 | 4668 | 1099 | 1125 | 948 | 958 | 938 | 948 | 954 | 952 | 1021 | 1027 |
| SERPENT-128 | 11767 | 11671 | 3108 | 2702 | 2855 | 2496 | 2842 | 2480 | 2847 | 2488 | 2868 | 2523 |
| SERPENT-192 | 13872 | 13852 | 3108 | 2705 | 2856 | 2478 | 2842 | 2465 | 2847 | 2467 | 2868 | 2505 |
| SERPENT-256 | 16073 | 15978 | 3108 | 2710 | 2857 | 2500 | 2842 | 2488 | 2847 | 2500 | 2868 | 2528 |
| TWOFISH-128 | 12907 | 12816 | 1063 | 1034 | 726 | 677 | 702 | 657 | 708 | 662 | 755 | 708 |
| TWOFISH-192 | 15311 | 15219 | 1061 | 1031 | 726 | 680 | 704 | 658 | 706 | 665 | 753 | 712 |
| TWOFISH-256 | 20706 | 20645 | 1061 | 1018 | 727 | 679 | 703 | 657 | 708 | 663 | 754 | 713 |

Cycles – Visual C 6.0 - 450 MHz Pentium II, 128MB RAM, Windows98

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 4937 | 4938 | 825 | 734 | 669 | 582 | 658 | 571 | 669 | 583 | 715 | 628 |
| MARS-192 | 4999 | 4999 | 825 | 734 | 669 | 578 | 658 | 572 | 667 | 582 | 716 | 629 |
| MARS-256 | 5175 | 5175 | 825 | 734 | 668 | 582 | 658 | 572 | 667 | 583 | 716 | 628 |
| RC6-128 | 2283 | 2284 | 638 | 622 | 339 | 327 | 321 | 310 | 330 | 320 | 379 | 354 |
| RC6-192 | 2408 | 2409 | 638 | 622 | 339 | 327 | 321 | 310 | 330 | 320 | 379 | 354 |
| RC6-256 | 2519 | 2520 | 638 | 622 | 339 | 327 | 321 | 310 | 330 | 320 | 379 | 354 |
| RIJNDAEL-128 | 1292 | 1722 | 987 | 987 | 810 | 801 | 808 | 789 | 826 | 796 | 894 | 866 |
| RIJNDAEL-192 | 2014 | 2553 | 1152 | 1135 | 987 | 969 | 983 | 957 | 1005 | 972 | 1079 | 1039 |
| RIJNDAEL-256 | 2594 | 3241 | 1329 | 1311 | 1161 | 1135 | 1158 | 1124 | 1173 | 1132 | 1238 | 1202 |
| SERPENT-128 | 6947 | 6935 | 1423 | 1262 | 1273 | 1116 | 1263 | 1107 | 1281 | 1122 | 1320 | 1162 |
| SERPENT-192 | 8857 | 8857 | 1423 | 1280 | 1274 | 1117 | 1263 | 1107 | 1281 | 1122 | 1320 | 1162 |
| SERPENT-256 | 10666 | 10683 | 1423 | 1256 | 1274 | 1117 | 1263 | 1108 | 1281 | 1122 | 1320 | 1162 |
| TWOFISH-128 | 9266 | 9249 | 1126 | 952 | 802 | 636 | 782 | 615 | 800 | 628 | 831 | 669 |
| TWOFISH-192 | 12707 | 12627 | 1130 | 952 | 802 | 634 | 782 | 616 | 795 | 622 | 832 | 673 |
| TWOFISH-256 | 17942 | 17863 | 1126 | 955 | 802 | 635 | 782 | 616 | 795 | 622 | 832 | 672 |

Cycles – Borland C++ 5.01 – 600 MHz Pentium III, 128MB RAM, Windows98

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 6833 | 6833 | 1143 | 1120 | 951 | 924 | 938 | 913 | 947 | 921 | 976 | 959 |
| MARS-192 | 7017 | 7017 | 1171 | 1131 | 951 | 926 | 938 | 914 | 940 | 917 | 980 | 959 |
| MARS-256 | 7245 | 7245 | 1143 | 1120 | 950 | 927 | 939 | 913 | 943 | 918 | 978 | 959 |
| RC6-128 | 5189 | 5186 | 1022 | 982 | 633 | 580 | 610 | 555 | 620 | 567 | 642 | 637 |
| RC6-192 | 5272 | 5271 | 1022 | 982 | 633 | 580 | 610 | 556 | 620 | 567 | 642 | 637 |
| RC6-256 | 5362 | 5363 | 1026 | 982 | 633 | 580 | 609 | 556 | 620 | 567 | 642 | 637 |
| RIJNDAEL-128 | 2213 | 2862 | 908 | 890 | 692 | 694 | 681 | 681 | 700 | 687 | 757 | 747 |
| RIJNDAEL-192 | 2981 | 3776 | 1031 | 1047 | 820 | 809 | 809 | 799 | 818 | 813 | 883 | 873 |
| RIJNDAEL-256 | 3727 | 4672 | 1152 | 1140 | 959 | 950 | 935 | 937 | 947 | 944 | 1002 | 996 |
| SERPENT-128 | 11850 | 11849 | 3161 | 2743 | 2859 | 2497 | 2842 | 2490 | 2855 | 2468 | 2870 | 2516 |
| SERPENT-192 | 13937 | 13916 | 3164 | 2739 | 2861 | 2484 | 2841 | 2467 | 2856 | 2495 | 2870 | 2536 |
| SERPENT-256 | 16133 | 16114 | 3165 | 2737 | 2859 | 2500 | 2841 | 2485 | 2849 | 2483 | 2869 | 2536 |
| TWOFISH-128 | 12938 | 12861 | 1085 | 1057 | 724 | 682 | 704 | 658 | 712 | 667 | 763 | 718 |
| TWOFISH-192 | 15347 | 15298 | 1085 | 1078 | 727 | 680 | 704 | 659 | 713 | 668 | 764 | 716 |
| TWOFISH-256 | 20760 | 20689 | 1085 | 1053 | 729 | 681 | 704 | 658 | 718 | 664 | 764 | 713 |

Cycles – Visual C 6.0 - 600 MHz Pentium III, 128MB RAM, Windows98

| | | | 1 block | | 16 blocks | | 128 blocks | | 1024 blocks | | 32768blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Ekey | Dkey | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| MARS-128 | 4934 | 4936 | 860 | 769 | 668 | 581 | 656 | 569 | 683 | 585 | 708 | 617 |
| MARS-192 | 4997 | 4997 | 860 | 769 | 668 | 578 | 656 | 569 | 682 | 585 | 709 | 618 |
| MARS-256 | 5171 | 5171 | 860 | 769 | 669 | 581 | 656 | 569 | 682 | 586 | 709 | 617 |
| RC6-128 | 2278 | 2279 | 672 | 657 | 339 | 327 | 318 | 307 | 325 | 318 | 366 | 346 |
| RC6-192 | 2403 | 2404 | 672 | 657 | 339 | 327 | 319 | 307 | 325 | 318 | 366 | 346 |
| RC6-256 | 2514 | 2515 | 672 | 657 | 339 | 327 | 319 | 307 | 325 | 318 | 366 | 346 |
| RIJNDAEL-128 | 1289 | 1724 | 1007 | 1006 | 811 | 802 | 805 | 784 | 824 | 794 | 880 | 848 |
| RIJNDAEL-192 | 2000 | 2553 | 1188 | 1169 | 987 | 966 | 981 | 955 | 1003 | 971 | 1069 | 1023 |
| RIJNDAEL-256 | 2591 | 3255 | 1365 | 1347 | 1160 | 1138 | 1155 | 1121 | 1171 | 1131 | 1227 | 1187 |
| SERPENT-128 | 6944 | 6933 | 1458 | 1315 | 1273 | 1113 | 1261 | 1104 | 1281 | 1120 | 1309 | 1150 |
| SERPENT-192 | 8853 | 8853 | 1459 | 1297 | 1273 | 1116 | 1260 | 1102 | 1281 | 1123 | 1309 | 1151 |
| SERPENT-256 | 10668 | 10668 | 1459 | 1315 | 1273 | 1115 | 1262 | 1103 | 1281 | 1120 | 1309 | 1150 |
| TWOFISH-128 | 9263 | 9241 | 1161 | 987 | 802 | 635 | 780 | 613 | 797 | 625 | 828 | 664 |
| TWOFISH-192 | 12722 | 12632 | 1165 | 987 | 802 | 633 | 779 | 613 | 791 | 619 | 828 | 666 |
| TWOFISH-256 | 17954 | 17876 | 1161 | 990 | 802 | 635 | 780 | 613 | 792 | 622 | 828 | 665 |

**5.2 Timing Tables**

Values in the tables are as follow:

- Ekey (time to make a key for encryption) is in Keys/sec;
- Encrypt (time to encrypt) is in Kbits/sec;
- Dkey (time to make a key for decryption) are in Keys/sec; and,
- Decrypt (time to decrypt) is in Kbits/sec.

GCC 2.8.1 - 200 MHz Pentium Pro, 64MB RAM, Linux

| | Ekey | Encrypt | Dkey | Decrypt |
|---|---|---|---|---|
| Mars-128 | 46729.0 | 39035.8 | 46511.6 | 37135.9 |
| Mars-192 | 44444.4 | 39035.8 | 44642.9 | 37135.9 |
| Mars-256 | 42918.5 | 38855.1 | 43103.4 | 37135.9 |
| RC6-128 | 59523.8 | 37300.9 | 58823.5 | 52454.4 |
| RC6-192 | 57142.9 | 37300.9 | 57803.5 | 52454.4 |
| RC6-256 | 56818.2 | 37300.9 | 57142.9 | 52454.4 |
| Rijndael-128 | 128205.1 | 42602.6 | 106383.0 | 41754.7 |
| Rijndael-192 | 88495.6 | 36175.4 | 74074.1 | 35562.3 |
| Rijndael-256 | 74074.1 | 31551.5 | 62500.0 | 30969.4 |
| Serpent-128 | 16891.9 | 13052.4 | 16920.5 | 16328.2 |
| Serpent-192 | 13123.4 | 13052.4 | 13140.6 | 16328.2 |
| Serpent-256 | 10559.7 | 13052.4 | 10582.0 | 16328.2 |
| Twofish-128 | 14471.8 | 20671.7 | 14450.9 | 22261.8 |
| Twofish-192 | 11086.5 | 20671.7 | 11025.4 | 22261.8 |
| Twofish-256 | 8305.6 | 20671.7 | 8291.9 | 22261.8 |

SGI 300 MHz R12000 w/4MB Cache, 512 MB RAM

| | GCC 2.8.1 | | | | MIPSpro C Compiler Version 7.30 | | | |
|---|---|---|---|---|---|---|---|---|
| | Ekey | Encrypt | Dkey | Decrypt | Ekey | Encrypt | Dkey | Decrypt |
| Mars-128 | 60975.6 | 63581.1 | 60975.6 | 66608.8 | 78125.0 | 67683.1 | 78125.0 | 71124.6 |
| Mars-192 | 59171.6 | 63581.1 | 59523.8 | 67141.6 | 76923.1 | 67683.1 | 76923.1 | 70526.9 |
| Mars-256 | 57803.5 | 63581.1 | 57803.5 | 66608.8 | 75188.0 | 67683.1 | 75188.0 | 70526.9 |
| RC6-128 | 147058.8 | 86522.7 | 147058.8 | 98737.7 | 166666.7 | 80699.1 | 166666.7 | 87424.0 |
| RC6-192 | 142857.1 | 86522.7 | 142857.1 | 98737.7 | 161290.3 | 80699.1 | 161290.3 | 87424.0 |
| RC6-256 | 138888.9 | 86522.7 | 138888.9 | 98737.7 | 156250.0 | 80699.1 | 156250.0 | 87424.0 |
| Rijndael-128 | 212766.0 | 58282.7 | 161290.3 | 58282.7 | 212766.0 | 74271.7 | 153846.2 | 79930.5 |
| Rijndael-192 | 163934.4 | 49080.1 | 125000.0 | 49368.8 | 142857.1 | 63103.0 | 109890.1 | 68233.4 |
| Rijndael-256 | 142857.1 | 42387.4 | 108695.7 | 42819.9 | 121951.2 | 54498.1 | 93457.9 | 58690.2 |
| Serpent-128 | 47393.4 | 42174.4 | 47393.4 | 46113.8 | 57471.3 | 42819.9 | 57471.3 | 45612.5 |
| Serpent-192 | 37878.8 | 41963.5 | 38022.8 | 46113.8 | 44247.8 | 42602.6 | 44247.8 | 45612.5 |
| Serpent-256 | 31250.0 | 41963.5 | 31250.0 | 46113.8 | 35461.0 | 42602.6 | 35461.0 | 45612.5 |
| Twofish-128 | 31055.9 | 59947.9 | 31055.9 | 63581.1 | 41493.8 | N/A | 41841.0 | N/A |
| Twofish-192 | 23255.8 | 60379.2 | 23310.0 | 64066.4 | 32786.9 | N/A | 33112.6 | N/A |
| Twofish-256 | 16420.4 | 59947.9 | 16447.4 | 63581.1 | 22321.4 | N/A | 22522.5 | N/A |

Sun 300 MHz UltraSPARC-II w/ 2MB Cache, 128 MB RAM

| | GCC 2.95 | | | | Sun Workshop Compiler 4.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Ekey | Encrypt | Dkey | Decrypt | Ekey | Encrypt | Dkey | Decrypt |
| Mars-128 | 48780.5 | 29867.3 | 48543.7 | 29242.9 | 52356.0 | 30081.4 | 53475.9 | 29973.9 |
| Mars-192 | 47393.4 | 29867.3 | 46948.4 | 29141.3 | 52356.0 | 30081.4 | 52083.3 | 30081.4 |
| Mars-256 | 46082.9 | 29867.3 | 45662.1 | 29242.9 | 51020.4 | 29973.9 | 51282.1 | 30081.4 |
| RC6-128 | 111111.1 | 20981.8 | 113636.4 | 20981.8 | 111111.1 | 20470.0 | 24390.2 | 20420.2 |
| RC6-192 | 108695.7 | 20981.8 | 108695.7 | 20981.8 | 101010.1 | 20520.1 | 24449.9 | 20470.0 |
| RC6-256 | 105263.2 | 20981.8 | 106383.0 | 20981.8 | 24390.2 | 20520.1 | 98039.2 | 20470.0 |
| Rijndael-128 | 172413.8 | 45612.5 | 131578.9 | 38498.6 | 166666.7 | 49368.8 | 117647.1 | 50864.9 |
| Rijndael-192 | 140845.1 | 37805.0 | 106383.0 | 32033.2 | 128205.1 | 41963.5 | 85470.1 | 43261.4 |
| Rijndael-256 | 117647.1 | 33042.1 | 90090.1 | 27517.1 | 108695.7 | 36490.0 | 73529.4 | 37467.4 |
| Serpent-128 | 30120.5 | 34537.9 | 30120.5 | 34969.6 | 33783.8 | 32156.0 | 33898.3 | 32912.6 |
| Serpent-192 | 25000.0 | 34255.9 | 25000.0 | 34969.6 | 27173.9 | 32033.2 | 27248.0 | 32912.6 |
| Serpent-256 | 21008.4 | 33841.5 | 21052.6 | 34824.5 | 22421.5 | 32156.0 | 22421.5 | 33042.1 |
| Twofish-128 | 22321.4 | 36972.3 | 22321.4 | 36020.2 | 21739.1 | 41963.5 | 21739.1 | 7851.0 |
| Twofish-192 | 16366.6 | 36972.3 | 16366.6 | 36020.2 | 16447.4 | 41754.7 | 16420.4 | 7880.5 |
| Twofish-256 | 11547.3 | 37300.9 | 11560.7 | 36020.2 | 12285.0 | 42174.4 | 12300.1 | 7865.7 |

Sun 2*360 MHz UltraSPARC-II w/ 4MB Cache, 256 MB RAM

| | GCC 2.95 | | | | Sun Workshop Compiler 4.2 | | | |
|---|---|---|---|---|---|---|---|---|
| | Ekey | Encrypt | Dkey | Decrypt | Ekey | Encrypt | Dkey | Decrypt |
| Mars-128 | 59523.8 | 36332.1 | 59523.8 | 35562.3 | 65359.5 | 36649.4 | 65359.5 | 36810.1 |
| Mars-192 | 57803.5 | 36175.4 | 57803.5 | 35412.3 | 64102.6 | 36649.4 | 64102.6 | 36810.1 |
| Mars-256 | 56179.8 | 36175.4 | 56179.8 | 35562.3 | 62500.0 | 36649.4 | 62500.0 | 36810.1 |
| RC6-128 | 138888.9 | 26227.2 | 138888.9 | 26227.2 | 142857.1 | 25587.5 | 142857.1 | 25587.5 |
| RC6-192 | 133333.3 | 26227.2 | 135135.1 | 26227.2 | 136986.3 | 25587.5 | 138888.9 | 25587.5 |
| RC6-256 | 129870.1 | 26227.2 | 129870.1 | 26227.2 | 131578.9 | 24978.3 | 131578.9 | 24978.3 |
| Rijndael-128 | 217391.3 | 55215.2 | 161290.3 | 47958.3 | 200000.0 | 59522.7 | 142857.1 | 61260.6 |
| Rijndael-192 | 172413.8 | 46886.6 | 129870.1 | 39965.3 | 158730.2 | 50864.9 | 107526.9 | 52454.4 |
| Rijndael-256 | 142857.1 | 40940.0 | 109890.1 | 34396.3 | 133333.3 | 44405.8 | 88495.6 | 45612.5 |
| Serpent-128 | 36101.1 | 41963.5 | 36231.9 | 42819.9 | 42372.9 | 39035.8 | 42372.9 | 39965.3 |
| Serpent-192 | 30303.0 | 41963.5 | 30303.0 | 42819.9 | 34013.6 | 39035.8 | 34013.6 | 39965.3 |
| Serpent-256 | 25641.0 | 41963.5 | 25641.0 | 42819.9 | 28328.6 | 39035.8 | 28328.6 | 39965.3 |
| Twofish-128 | 27322.4 | 45122.1 | 27248.0 | 43039.5 | 26738.0 | 53118.4 | 26738.0 | 51489.0 |
| Twofish-192 | 20080.3 | 44880.8 | 20080.3 | 43039.5 | 20120.7 | 53456.7 | 20120.7 | 51489.0 |
| Twofish-256 | 14184.4 | 44880.8 | 14164.3 | 43261.4 | 15015.0 | 53456.7 | 15037.6 | 51806.8 |

# 6. Conclusions

## 6.1 PC

Due to the testing mechanisms used in obtaining data, the most reliable and accurate values obtained for performance measurement of the candidate algorithms are the cycle counting measurements on the PC. Additionally, cycle count values for encryption and decryption were obtained for various data block lengths. These values provide interesting results. For the most part, once the data length was greater than one block (128 bits), the encryption and decryption speeds were consistent within each algorithm. For this reason, NIST focused on the message block length of 128 blocks (2046 bytes), which is a typical size for an electronic mail message. The fastest algorithm for key setup on the PC platform is Rijndael for all compiler and PC hardware/software configurations, followed closely by RC6 and then Mars. Serpent and Twofish are considerably slower than the other algorithms for key setup time. Encryption speed had more variability across compiler and hardware/software platforms. RC6 tends to fall near the top of PC encryption speed followed by Mars, Twofish, and Rijndael. Serpent is consistently at the bottom of the list for encryption speed.

Brian Gladman [4] has performed similar efficiency experiments, the results of which are available on a web page he maintains. The tests that Gladman conducted used code that he developed independently from the submitters' code. Gladman's results are similar to those listed above. Gladman's results for key setup time have the algorithms in basically the same order. The exception being the fact that Serpent's key setup time was greatly improved and ahead of Mars. Again, for encryption speed, Gladman's results coincide with the ordering of the algorithms listed above.

## 6.2 Sun

The UltraSPARC™ CPU found in the Sun systems on which testing was performed did not allow access to a cycle count mechanism. Performance numbers on these systems are based on the Timing Test Program. Two different compilers were used on the Sun. The data from both these compilers yielded similar results. The fastest algorithms with respect to encryption speed are Rijndael and Twofish, followed by Serpent and Mars, and finally by RC6. However, with respect to key setup Rijndael and RC6 are the fastest followed by Mars which is separated by a wide margin. Serpent and Twofish are last after another wide margin.

Helger Lipmaa reports very similar results on an UltraSPARC-II platform [5]. Lipmaa's table only reports encryption speed. The most noticeable difference is that on his table, the value for the encryption speed of RC6 is closer to those for Mars and Serpent.

## 6.3 SGI

The SGI system provides another 64-bit processor running the same version of the GCC compiler used for the Sun testing described in Section 6.2. Additionally, the MIPSpro compiler provided another configuration for comparison. The results for these compilers place RC6 as the fastest algorithm for encryption by a wide margin, followed by Mars, Twofish, Rijndael and

Serpent. For key setup, RC6 and Rijndael are the fastest, followed by Mars, Serpent, and Twofish, which are separated by a wide margin.

### 6.4 Overall Performance

The consistent top performers across all platforms with respect to key setup are Rijndael and RC6. Serpent and Twofish are usually significantly poorer performers; however, Gladman reports a much better value for Serpent key setup, placing Serpent ahead of Mars. Encryption speed values tend to vary much more depending on the platform being analyzed. Rijndael, Mars, and Twofish have the most even encryption performance across platforms – not always the fastest, but never near the bottom of the pack. RC6, on the other hand, was the slowest on the Sun systems but the fastest on the SGI and very nearly the fastest on the PC. Serpent is typically the slowest or towards the bottom of the list on encryption speed across platforms.

## 7. References

[1] "Request for Comments on the Finalist (Round 2) Candidate Algorithms for the Advanced Encryption Standard (AES)," Federal Register, Volume 64, Number 178, pp. 50058-50061, Sept. 15, 1999.

[2] "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," Federal Register, Volume 62, Number 177, pp. 48051-48058, Sept. 12, 1997.

[3] "Using the RDTSC Instruction for performance monitoring," http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM, Intel Corporation, 1997.

[4] Brian Gladman, "AES Second Round Implementation Experience," http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/index.htm, March 1999.

[5] Helger Lipmaa, "AES Ciphers: Speed," http://home.cyber.ee/helger/aes/table.html, 1999.

# NIST Performance Analysis of the Final Round Java™ AES Candidates

Jim Dray
Computer Security Division
The National Institute of Standards and Technology
james.dray@nist.gov

March 15, 2000

## 1. Introduction

NIST solicited candidate algorithms for the Advanced Encryption Standard (AES) in a Federal Register Announcement dated September 12, 1997[1]. Fifteen of the submissions were deemed "complete and proper" as defined in the Announcement, and entered the first round of the AES selection process in August 1998. Since that time, NIST has been working with a worldwide community of cryptographers to evaluate the submissions according to the criteria established in[1]. Five candidates were subsequently chosen to enter the final round of the selection process: MARS, RC6, Rijndael, Serpent, and Twofish.

A previous NIST publication entitled "Report on the NIST Java™ AES Candidate Algorithm Analysis"[2] documents the first round analysis performed by NIST, using the Java Development Kit (JDK) Version 1.1.6. Only IBM has submitted official modifications to their candidate (MARS) prior to the final round. Results of the first round analysis using the JDK1.1.6 are therefore still valid for the other four candidates. The revised version of MARS was tested under both JDK1.1.6 and JDK1.3, to ensure an accurate comparison of the modified algorithm's performance in both environments. Performance data for 128, 192, and 256-bit keysizes are also included in the second round analysis.

The JDK itself has gone through two major revisions since the first round. This paper documents additional performance data for the five AES finalists obtained under JDK1.3, and should be used in combination with the first round NIST Java AES analysis to obtain a complete picture of the characteristics of the finalists in different Java environments. Some background information from the first round analysis is repeated herein for convenience. Comments should be addressed to the author at the email address above.

## 2. Java Platform

AES candidate algorithm submitters were required to provide optimized implementations of their algorithms in Java and the C language. The rationale for this was to provide more information than could be obtained by testing implementations in a single language, and to take advantage of the hardware independence of the Java virtual machine.

The Java virtual machine presents a uniform abstraction of the underlying hardware platform to a Java application or applet. A Java programmer compiles source code into byte code files, which are then interpreted by the Java virtual machine at runtime (byte code files are also known as class files). In theory, a Java byte code file can be interpreted on any hardware platform running the Java virtual machine without recompilation. Since the virtual machine isolates the Java programmer from the underlying hardware, Java programmers cannot write machine-specific code to take advantage of the unique features of a particular platform. Machine-specific code allows for optimization on a given computing platform, but also eliminates the code portability that is a cornerstone of the Java philosophy.

The Java environment has two characteristics that facilitate the AES evaluation process. First, candidate algorithms written in Java can be easily moved from one platform to another to compare performance on different processors at different system clock speeds. Second, submitters cannot write machine-specific code and so all implementations are on a level playing field.

Java does not provide the level of performance that can be attained in some other languages (C or assembler, for example). However, many applications do not require high-speed encryption of large amounts of data, and cryptoalgorithms implemented in Java are easier to integrate into Java applications. Other languages and hardware implementations will be used for applications where absolute performance is an issue, but there will also be a broad range of applications where the ease of implementing, integrating, and maintaining Java AES code outweighs the performance issue.

## 3. Evaluation Criteria

The NIST Java AES evaluation process is designed to directly address the criteria published in the Federal Register Announcement[1], Section 4. The goal is to provide objective results that can be clearly quantified for use in the selection process. Sections of the Announcement that describe selection critera relevant to the Java AES analysis are repeated here for convenience:

***COST***

ii. *Computational Efficiency:* "…Computational efficiency essentially refers to the speed of the algorithm. NIST's analysis of computational efficiency

will be made using each submission's mathematically optimized implementations on the platform specified under Round 1 Technical Evaluation below."

iii. *Memory Requirements:* "Memory requirements will include such factors as gate counts for hardware implementations, and code size and RAM requirements for software implementations."

***ALGORITHM AND IMPLEMENTATION CHARACTERISTICS***

*i. Flexibility:*

b. "The algorithm can be implemented securely and efficiently in a wide variety of platforms and applications (e.g. 8-bit processors, ATM networks, voice & satellite communications, HDTV, B-ISDN, etc.)."

ii. *Simplicity:* "A candidate algorithm shall be judged according to relative simplicity of design."

Additionally, in Section 6.B (**Round I Technical Evaluation**):

iii. *Efficiency testing:* "Using the submitted mathematically optimized implementations, NIST intends to perform various computational efficiency tests for the 128-128 key-block combination, including the calculation of the time required to perform:

- Algorithm setup,
- Key setup,
- Key change, and
- Encryption and decryption.

NIST may perform efficiency testing on other platforms."

In condensed form, the published NIST criteria require testing of speed for a set of cryptographic operations, code size and RAM requirements, flexibility, and simplicity of design. Since the candidates have been implemented in Java, flexibility is a given for the reasons discussed in the previous section. The Java AES candidates will run on any device containing a Java virtual machine and adequate memory, although performance will obviously vary depending on the processing power of the underlying hardware.

# 4. Test Procedures

## 4.1 Overview

The test results presented here were obtained from the NIST-specified hardware platform and the most recent version of the Java environment available at the time of this writing (JDK1.3, beta release). Results for other hardware/Java virtual machine combinations will be made available on the AES home page at http://www.nist.gov/aes, and in papers submitted to NIST by other organizations[3,4,5]. Detailed test results are presented in tabular form in Appendices A and B, and chart form in Appendix C. All NIST testing was performed through the Applications Programming Interface (API) specified in the NIST/Cryptix Java AES Toolkit. Links to the Toolkit and the Java AES API specification can be found at http://csrc.nist.gov/encryption/aes/earlyaes.htm.

The Java compiler provided with JDK1.3 accepts a command line code optimization switch (-O). However, the JDK1.3 documentation[6] states that this switch "does nothing in the current implementation". Presumably the compiler accepts the optimization switch for reasons of backward compatibility.

### 4.2 Procedures

Candidate algorithms were compiled from source files provided by submitters using the JDK1.3 compiler. The resulting bytecode files were packaged into a standard Java ARchive (JAR) file named AESCLASSES.jar.

A Java application was developed to allow testing of any candidate/ keysize/operation combination. The test application instantiates the desired candidate from AESCLASSES.jar, and uses the Java reflection API to invoke the Basic API methods.

500,000 cycles of each candidate/keysize/crypto operation were executed, and the total time was recorded for each combination. Start and stop times were obtained through calls to the System.time.millis() method provided in the Java core library, immediately before and after starting the loop that executed the crypto operations. Charts 1, 2, and 3 present performance data for key setup, encrypt, and decrypt operations, respectively. Data points are included for 128, 192, and 256-bit key sizes. For the majority of candidates, encryption and decryption speed is approximately equal for all three key sizes. Rijndael is a minor exception: encryption speed decreases by approximately three percent for each stepwise increase in key size.

## 5. Results

In comparison to the JDK1.1.6 performance data presented in NIST's previous paper[2], the results obtained with JDK1.3 show a striking increase in execution speed for all candidates. On average, the five candidates perform 128-bit key setup operations eleven times faster. The average speed for encrypt and decrypt operations has increased by a factor of five. The same hardware platform and program code (except for MARS) were used for both first round and final round testing, so the overall increase in performance

can be attributed to differences in the Java environment. In particular, JIT (Just-In-Time) compilation was not used during the first round performance analysis due to a bug in the JDK1.1.6 JIT implementation that caused problems with certain candidates. Usage of the JIT compiler under JDK1.1.6 increases performance by a factor of ten for most Java programs.

Performance data for the new version of MARS under JDK1.1.6 are presented separately in Appendix A. The test setup for the MARS/JDK1.1.6 analysis was exactly the same as for the other algorithms during the first round, and is described in[2].

In addition to the overall performance increase of the finalists under JDK1.3, there were some changes in the relative ordering of candidates. Most of these changes in order were due to relatively small performance differences, as shown in Appendices B and C. The results for 128-bit keysize operations are summarized below, with candidates ordered from fastest to slowest:

128-bit Key Setup:

JDK1.1.6: Rijndael, RC6, MARS, Twofish, Serpent

JDK1.3: RC6, MARS, Rijndael, Serpent, Twofish

128-bit Encrypt:

JDK1.1.6: RC6, Rijndael, MARS, Serpent, Twofish

JDK1.3: Rijndael, RC6, MARS, Serpent, Twofish

128-bit Decrypt Operations:

JDK1.1.6: RC6, Rijndael, MARS, Serpent, Twofish

JDK1.3: Rijndael, RC6, MARS, Twofish, Serpent

# REFERENCES


1. "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)", Federal Register: September 12, 1997 (Volume 62, Number 177), Pages 48051-48058.

2. J. Dray, "Report on the NIST Java™ AES Candidate Algorithm Analysis", http://csrc.nist.gov/encryption/aes/round1/r1-java.pdf , November 8, 1999.

3. A. Folmsbee, "AES Java™ Technology Comparisons", Proceedings of the Second Advanced Encryption Standard Candidate Conference, March 22, 1999, Pages 35-50.

4. K. Aoki, "Java Performance of AES Candidates", Submitted to NIST via email in response to the call for public comments on the AES candidates, April 15, 1999.

5. A. Sterbenz, P. Lipp, "Performance Analysis of Java Implementations of the Second Round AES Candidate Algorithms", Submitted to NIST via email, January 2000.

6. Java™ 2 SDK, Standard Edition Documentation (Version 1.3), http://java.sun.com/products/jdk/1.3/docs/.

# APPENDIX A: JDK1.1.6 DATA FOR MARS

| Key Size | Key Setup | Encrypt | Decrypt |
|---|---|---|---|
| 128 bits | 165 | 462 | 444 |
| 192 bits | 244 | 466 | 444 |
| 256 bits | 324 | 465 | 445 |

Table data are presented in kilobits per second.

# APPENDIX B: RAW DATA TABLES

| Algorithm | setKey128 | setKey192 | setKey256 |
|---|---|---|---|
| RC6 | 2233 | 3335 | 4444 |
| MARS | 2110 | 3131 | 4131 |
| Rijndael | 1191 | 1574 | 1733 |
| Serpent | 487 | 734 | 979 |
| Twofish | 286 | 327 | 361 |

| Algorithm | Encrypt128 | Encrypt192 | Encrypt256 |
|---|---|---|---|
| Rijndael | 4855 | 4664 | 4481 |
| RC6 | 4698 | 4740 | 4733 |
| MARS | 3738 | 3707 | 3733 |
| Serpent | 1843 | 1855 | 1861 |
| Twofish | 1749 | 1749 | 1744 |

| Algorithm | Decrypt128 | Decrypt192 | Decrypt256 |
|---|---|---|---|
| Rijndael | 4819 | 4624 | 4444 |
| RC6 | 4733 | 4698 | 4740 |
| MARS | 3965 | 3965 | 3936 |
| Serpent | 1873 | 1897 | 1896 |
| Twofish | 1781 | 1775 | 1781 |

Table data are presented in kilobits per second.

# APPENDIX B: PERFORMANCE DATA CHARTS

**Chart 1: Key Setup**

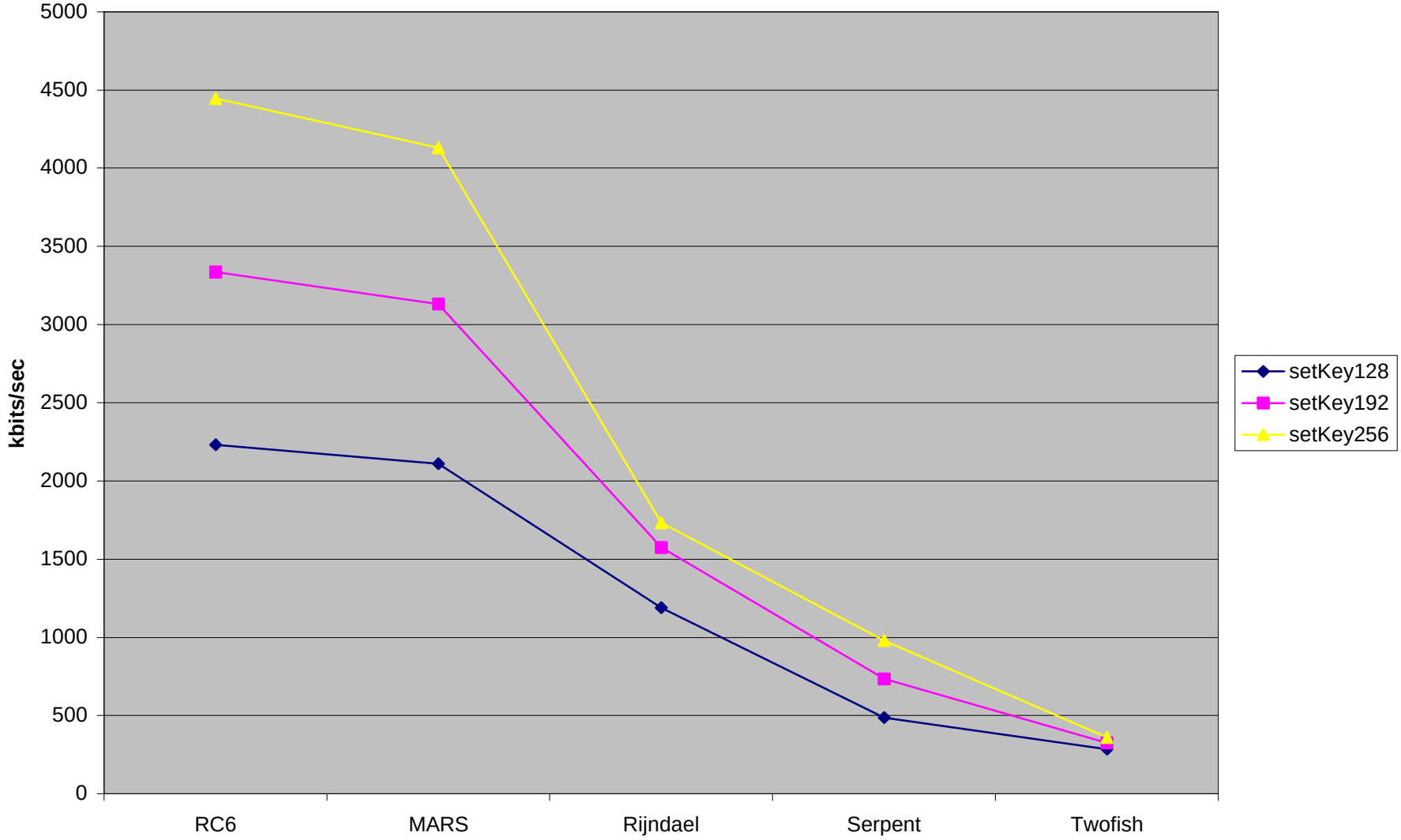


**Chart 2: Encrypt**

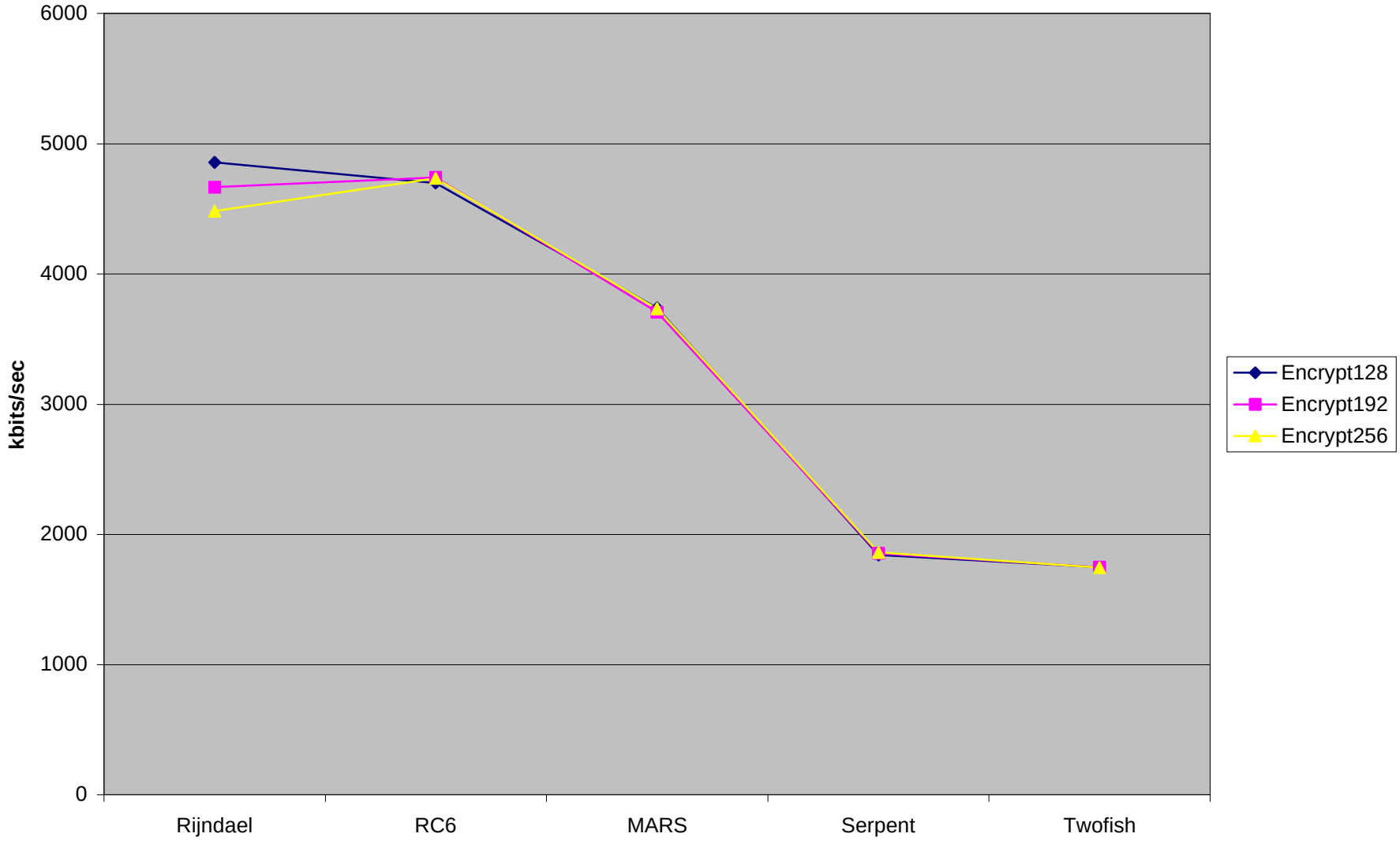
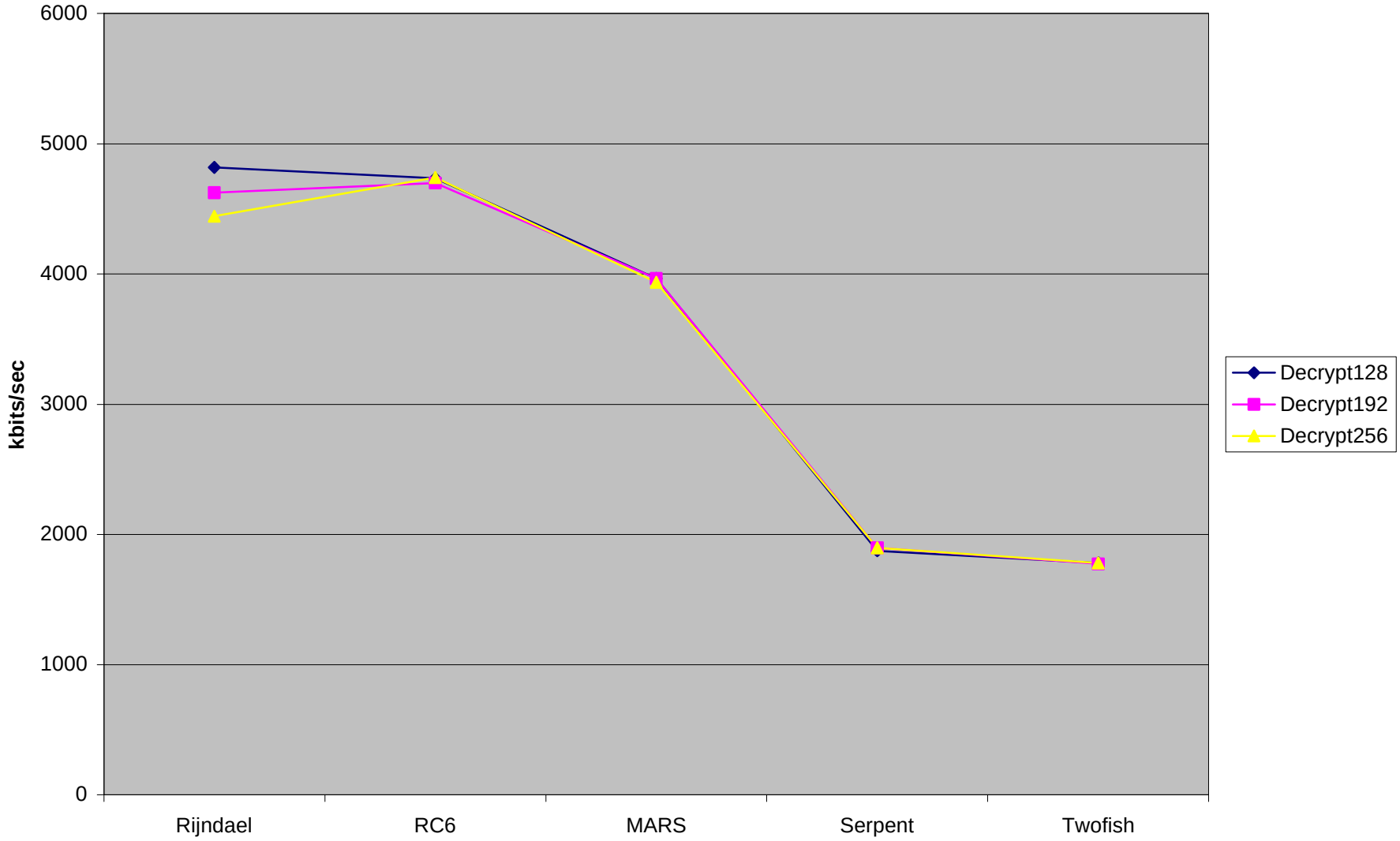


**Chart 2.3: Decrypt**



# Performance of the AES Candidate Algorithms in Java

Andreas Sterbenz, Andreas.Sterbenz@iaik.at
Peter Lipp, Peter.Lipp@iaik.at
Institute for Applied Information Processing and Communications
Graz, University of Technology
Inffeldgasse 16, A-8010 Graz, Austria
http://www.iaik.at

## Abstract

*We analyze the five remaining AES candidate algorithms MARS, RC6, Rijndael, Serpent, and Twofish as well as DES, Triple DES, and IDEA by examining independently developed Java implementations. We give performance measurement results on several platforms, list the memory requirements, and present a subjective estimate for the implementation difficulty of the algorithms. Our results indicate that all AES ciphers offer reasonable performance in Java, the fastest algorithm being about twice as fast as the slowest.*

## 1. Introduction

The performance of the AES candidates has been the subject of significant discussion, both in the authors' specifications as well as by other parties. Most of this discussion was focused on C and assembler implementations. Some attention has been given to Java implementations but the results were not fully conclusive. This was mostly caused by the fact that the authors' reference Java implementations were evaluated which vary significantly in their coding assumptions and in the degree to which they were subject to optimizations. We intend to fill this gap by evaluating independently developed, consistent Java implementations and comparing the AES candidates' performance to ciphers currently in use.

## 2. Implementation Notes

The code was developed at the IAIK by Andreas Sterbenz. The AES core code is available under a free license including source at [1] or with a JCE 1.2 compatible API as part of the IAIK JCE library. Serpent S-Box expressions and Rijndael and Twofish setup code are based on C code developed by Dr. Brian Gladman [2].

The design paradigm used is derived from the Java Cryptography Extension (JCE) defined by Javasoft and modified for use within the IAIK JCE library: for each cipher stream a Java object is created which is then initialized with a certain key in either encryption or decryption mode. Then the data to be encrypted is passed to the encrypt (decrypt) method one 128 bit block at a time. Buffering, block chaining, and padding are all performed on a higher level and do not influence the design of the core code. Therefore, for each AES cipher only three methods need to be provided: key setup, encryption, and decryption.

The algorithms have been subject to significant optimization work. The primary focus for the optimization was to maximize encryption and decryption throughput. Secondary and tertiary goals were key setup speed and memory usage, respectively.

## 3. Java

The Java programming language has become fairly popular in recent years. This is partly due to the fact that Java programs are platform (i.e. processor and operating system) independent in both source and binary form. This is possible by employing a compilation model different from that in most other languages. Instead of compiling source code into machine code for one particular processor family, the compiler produces machine code (called "bytecode") for an imaginary Java Virtual Machine (JVM). At runtime this bytecode is then translated into machine code by a JVM implementation for the particular platform.

This extra step has influences on the programming process when optimizing code. It takes you one step farther away from the hardware making some typical optimization tricks impossible, like for example directly using the processor rotation instruction. Another problem is that a sizable portion of the compilation is delayed until runtime and performed by the JVM. As they are not designed for optimizations this has the effect that those optimizations are not made.

Of course there are several options for the translation of bytecode to machine code. The simplest and most obvious is to use an interpreter: take one JVM instruction at a time and execute the corresponding machine code instruction(s). Much better performance is offered by so-called Just-In-Time (JIT) compilers. They take an entire method and translate it to machine code prior to its first execution, subsequently

the generated machine code is executed. JITs are now the common JVM type on most platforms and offer an approximately ten times performance improvement over interpreters. As a third type of JVMs there are hybrid variants aimed at reducing the initial delay caused when the JIT compilers translate a large number of methods at program startup, but this is not relevant for our application.

### 3.1. Java in Cryptographic Applications

Today the opinion that Java is not the language to be used for cryptographic applications still seems to be popular. Obviously we do not agree. While Java is of course slower than C the difference is typically less than a factor of two, heavily optimized C code excluded, as demonstrated by the results presented in this paper. Although this difference is of course significant Java on today's hardware is faster than C on two year old hardware. The point being that while Java will hardly be the language of choice for high load servers it may well be the choice for medium load servers and especially clients. Add to that handheld and other small devices and performance in Java becomes an issue.

One particular advantage of Java is that there is a well established standard cryptographic API, the JCA and JCE architecture from Javasoft. The success of cryptography libraries in Java including the libraries from the IAIK confirms this position.

## 4. Evaluation Parameters

The algorithms were implemented in Java. Those implementations were evaluated with respect to three criteria: execution speed, memory usage, and implementation difficulty.

### 4.1. Execution Speed

For symmetric ciphers there are three components that make up the time required to encrypt some data: static initialization time, key setup time, and data encryption time.

Static initialization is used to perform certain preparation steps, generate constant tables, etc. Because it takes very little time and is largely dependent on the code size vs. speed tradeoff chosen in the implementation it was not measured.

Key setup is used to initialize a cipher for a certain key, i.e. perform round key generation, etc. It is performed once per encryption stream. It may be dependent on whether encryption or decryption mode is chosen and on the key length. For the ciphers analyzed only Rijndael and IDEA have different key setup times for encryption and decryption modes and only Rijndael and Twofish significantly different setup times for different key lengths.

Data encryption time is of course the time it takes to encrypt data bits once the cipher has been properly initialized. The AES candidates are 128 bit block ciphers, that means one encryption operation is performed every 16 data bytes. Again it may vary with the cipher's mode and key length. For all ciphers analyzed the encryption and decryption times are virtually identical and only Rijndael's performance is dependent on the key length.

#### 4.1.1. Key Setup Speed Measurement

Key setup speed was determined as described by the following pseudo code:

```
Repeat 128 times
  Generate 32 random keys
  Start timer
  For each key
    Repeat 1024 times
      Initialize cipher with key
  Stop timer
```

To obtain the final value the average of all measurements within three standard deviations was calculated.

#### 4.1.2. Encryption Speed Measurement

Similarly encryption speed was measured:

```
Repeat 128 times
  Generate a random key
  Initialize cipher with key
  Start timer
  Repeat 2048 times
    Encrypt a 1024 byte array
  Stop timer
```

The same method as above was used to obtain the final value. Note that the same 1024 byte array is encrypted each time which takes full advantage of the CPU caches. In other words, the results presented here are upper boundaries for real world performance.

#### 4.1.3. Environment

The code was compiled using Symantec Visual Cafe 2.5a with optimizations enabled. The results were obtained by running the tests on a machine with an Intel Pentium Pro 200 MHz CPU and 128 MB RAM running Windows NT 4.0 with Service Pack 4. Performance wise this is virtually identical to the NIST reference platform (64 MB RAM and running Windows 95).

However, it should be noted that the actual development and optimization was done on a machine using an AMD K6-2 processor. The optimizing process, which includes trial and error strategies was performed to maximize throughput on this machine and not the reference machine. This may in some cases

| | DES | Triple DES | IDEA | MARS | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|---|---|---|
| **Class File Size** | n/a | n/a | n/a | 9984 | 1931 | 4900 | 12483 | 5204 |
| **Per process memory** | 5120 | 5120 | 0 | 3220 | 0 | 20520 | 0 | 6816 |
| **Per instance memory** | 128 | 384 | 416 | 220 | 432 | 240 | 576 | 4400 |

**Table 1: Class file size and memory usage in bytes.**

lead to cases were the performance on the reference machine is not as good as it could be.

### 4.2. Memory Usage

We give an estimate of the memory required for each of the algorithms. The size of the class file (debugging information removed) is listed to give an idea of the total size, consisting of code size and data like S-Box tables, etc. This is only done for the AES candidates because the other algorithms use a slightly different API which would skew results.

Probably more interesting is the amount of memory required during execution. We list the data memory used obtained by counting the variables used in the source code. Overhead for arrays or data allocated on the stack is not counted as it is fairly small and approximately identical for all of the algorithms.

### 4.3. Implementation Difficulty

We also assign implementation difficulty "grades" to the algorithms. In difference to the other criteria these were not measured but are subjective estimates for the time it required to arrive at an acceptably fast implementation of the algorithm. If we want to look at it in a quasi formal way we identify the following factors:

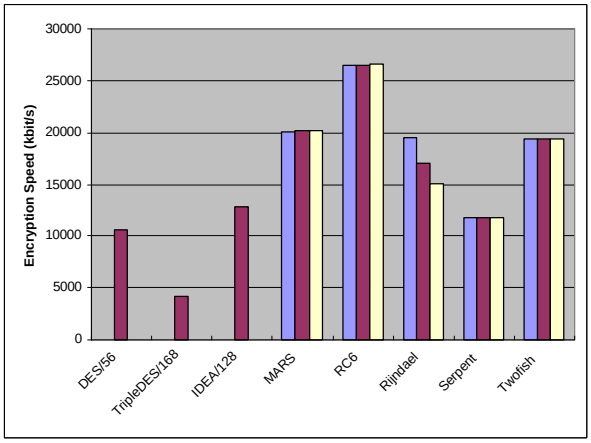
- Time taken to understand the algorithm (at least well enough to be able to implement it).
- Time taken to understand how to efficiently implement the algorithm on a 32 bit platform. As some algorithms need to be coded very differently from their specification in order to be efficient this part may constitute a significant part of the total time.
- Time taken to actually code the implementation.

The first two points are of course to some degree dependent on the documentation provided by the algorithm designers and other parties. Therefore, new or improved documentation may update the results given here.

## 5. Algorithms

### 5.1. DES

The Data Encryption Standard (DES) is the current US standard which the AES will eventually replace. It dates back from the 1970s and has become inadequate in particular because of its key length of only 56 bit. DES was designed for hardware implementations and requires tricks to operate moderately fast in 32 bit software implementations. These tricks are not obvious which is why DES only earns a B- for implementation difficulty. However, an advantage of DES over all other algorithms examined except Triple DES is that the encrypt and decrypt operations are identical save for the key schedule resulting in smaller code.




### 5.2. Triple DES

Triple DES overcomes the limitation of the short DES key length by using three DES cores with separate keys in sequence. This results in an effective strength of 112 bit (meet in the middle attacks) at the price a significant performance drop. Triple DES only performs somewhat faster than one third of the speed of DES (reduced overhead, leaving off the initial and final permutations), which means it is very slow in software. Implementation difficulty is B- as with DES.

### 5.3. IDEA

IDEA is a 16 bit oriented cipher which uses multiplication modulo 65537 for fast diffusion. Consequently it performs quite well compared to DES (depending on the processors multiplication unit). However, its key setup is quite slow in decryption mode as multiplicative inverses have to be calculated. It has also to be noted that a class of weak keys has been discovered. For implementation difficulty it earns B+ as that is fairly straight forward.

### 5.4. MARS

MARS is the first of the AES candidates we examine. It uses 8 rounds of unkeyed mixing before and after the core encryption rounds. One of its advantages is that a 32 bit implementation can be written exactly the way the algorithm is specified, also aided by the pseudo code given in the specification. Implementation difficulty B+.

### 5.5. RC6

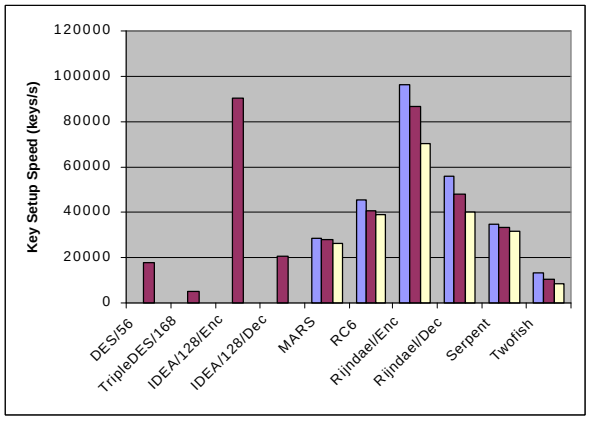
RC6 is a cipher that evolved from RC5. It is very simple to understand and implement and very fast on 32 bit processors; implementation difficulty A. Although the least time was spent on optimizing RC6 it still comes out as the fastest algorithm on almost all platforms.

### 5.6. Rijndael

Rijndael was designed based on strong mathematical foundations. Implemented on 32 bit processors only table lookup, XOR and shift operations are used. The number of rounds in the Rijndael cipher increases with the key length resulting in decreasing speed for both key setup and encryption. Key setup for Rijndael is very fast for in encryption mode but slower in decryption mode as an additional inversion step is required. Implementation difficulty B.

### 5.7. Serpent

Serpent was designed for so-called bitslice implementations. The idea is to view a 32 bit register as 32 one bit registers which are operated on by 32 one bit SIMD processors with e.g. logical operations. However, S-Boxes have to be implemented via logical expressions in this mode. Efficient expressions are not trivial to obtain and no expressions are given in the specification, contributing to the B- grade for implementation difficulty. Serpent was designed with a large safety margin of 32 rounds vs. about 20 minimum secure rounds. This results in lower speed, the




penalty depending on the JVM implementation and the processor.

### 5.8. Twofish

Twofish is a very flexible cipher that allows for several implementation options allowing a memory usage vs. key setup speed vs. encryption speed tradeoff. As maximum encryption throughput was desired the "full keying" option was chosen for the implementation. A special property of Twofish is that key dependent S-Boxes are used. This somewhat hurts performance on certain JVMs, in particular when using the Symantec JIT compiler that comes with the JDK on the Windows platform and which was used for the measurements. This means that Twofish may be somewhat faster compared to the other algorithms on other platforms. As Twofish is a quite complicated cipher it earns B- for implementation difficulty.

## 6. Conclusions

We have analyzed the performance of the AES candidate and other ciphers.

The results for encryption and decryption speed show that RC6 is about 25% faster than the other algorithms. Then MARS, Rijndael, and Twofish follow with virtually identical performance for 128 bit keys, Rijndael being slower for longer keys. Serpent is trailing behind but is still about as fast as IDEA. DES follows with Triple DES far behind. These results are similar to some tests made using C implementations but deviate much from Java studies. The results also show that Java is no more than a factor of 2-3 slower than optimized C code.

The key setup performance is more varied with the fastest AES candidate more than 7 times as fast as the slowest. This appears to be partly due to differing opinions about the purpose of the key schedule. It could be viewed as a one way hash function: accepting an arbitrarily long key, producing output of fixed length (the round keys). All round keys depend on all input bits and obtaining a round key (using some

attack) does not yield any information about the original key. Some algorithms try to approximate this ideal while others only generate the necessary key material in a straight forward way.

In any case Rijndael is the fastest algorithm with respect to key setup, although it is not that far ahead when keys longer than 128 bit are used and in decryption mode. Twofish has a fairly slow key setup using this implementation option.

In summary it can be said that if properly implemented all algorithm offer reasonable performance in Java. The results are mostly in line with those obtained by studies evaluating C implementations.

## 7. References


[1] IAIK. *The IAIK AES home page* http://jcewww.iaik.at/aes/

[2] Brian Gladman. *AES Implementations in C* http://www.btinternet.com/~brian.gladman/cryptography_technology/aes2/aes.r2.algs.zip

[3] X. Lai, J.L. Massey and S. Murphy. *Markov ciphers and differential cryptanalysis* Advances in Cryptology, Proceedings Eurocrypt'91, LNCS 547, D.W. Davies, Ed., Springer-Verlag, 1991, pp. 17-38.

[4] Jim Dray. *Report on the NIST Java AES Candidate Algorithm Analysis* Available from http://csrc.nist.gov/encryption/aes/round2/round2.htm

[5] Lawrence E. Bassham III. *Efficiency Testing of ANSI C Implementations of Round1 Candidate Algorithms for the Advanced Encryption Standard* http://csrc.nist.gov/encryption/aes/round2/round2.htm

[6]


## Appendix

This appendix includes the full performance figures as obtained on the reference machine.

| Encryption Speed (kbit/s) | DES (56 bit) | Triple DES (168 bit) | IDEA | MARS | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|---|---|---|
| **128 bit key** | 10508 | 4178 | 12820 | 19718 | 26212 | 19321 | 11464 | 19265 |
| **192 bit key** | n/a | n/a | n/a | 19760 | 26192 | 16922 | 11474 | 19296 |
| **256 bit key** | n/a | n/a | n/a | 19737 | 26209 | 14957 | 11471 | 19275 |

| Decryption Speed (kbit/s) | DES (56 bit) | Triple DES (168 bit) | IDEA | MARS | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|---|---|---|
| **128 bit key** | 10519 | 4173 | 13018 | 19443 | 24338 | 18868 | 11519 | 18841 |
| **192 bit key** | n/a | n/a | n/a | 19670 | 24382 | 16484 | 11514 | 18841 |
| **256 bit key** | n/a | n/a | n/a | 19489 | 24279 | 14468 | 11533 | 18806 |

| Encryption Key Setup (keys/s) | DES (56 bit) | Triple DES (168 bit) | IDEA | MARS | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|---|---|---|
| **128 bit key** | 18128 | 5150 | 90571 | 28680 | 45603 | 96234 | 34729 | 13469 |
| **192 bit key** | n/a | n/a | n/a | 27928 | 40625 | 86773 | 33516 | 10556 |
| **256 bit key** | n/a | n/a | n/a | 26683 | 29069 | 70494 | 31973 | 8500 |

| Decryption Key Setup (keys/s) | DES (56 bit) | Triple DES (168 bit) | IDEA | MARS | RC6 | Rijndael | Serpent | Twofish |
|---|---|---|---|---|---|---|---|---|
| **128 bit key** | 18039 | 5136 | 20737 | 28743 | 45709 | 56017 | 34687 | 13469 |
| **192 bit key** | n/a | n/a | n/a | 27917 | 40625 | 48324 | 33560 | 10550 |
| **256 bit key** | n/a | n/a | n/a | 26731 | 39028 | 39963 | 31973 | 8531 |

# *Session 4:*

# *"Cryptographic Analysis and Properties"*

# *(I)*

# MARS Attacks! Preliminary Cryptanalysis of Reduced-Round MARS Variants

John Kelsey and Bruce Schneier

Counterpane Internet Security, Inc., 3031 Tisch Way, San Jose, CA 95128
{kelsey,schneier}@counterpane.com

**Abstract.** In this paper, we discuss ways to attack various reduced-round variants of MARS. We consider cryptanalysis of two reduced-round variants of MARS: MARS with the full mixing layers but fewer core rounds, and MARS with each of the four kinds of rounds reduced by the same amount. We develop some new techniques for attacking both of these MARS variants. Our best attacks break MARS with full mixing and five core rounds (21 rounds total), and MARS symmetrically reduced to twelve rounds (3 of each kind of round).

## 1 Introduction

MARS [BCD+98] is a block cipher submitted by IBM to the AES [NIST97a] [NIST97b], and one of the five finalists for AES. The cipher has an unconventional structure, consisting of a cryptographic "core" in the middle, and a "wrapper" surrounding the core to protect it from various kinds of attack. As with all ciphers, the only way we know to determine the strength of MARS is to try to cryptanalyze various weakened versions of it.

In this paper, we discuss attacks on reduced-round variants of MARS. Because of MARS' unconventional structure, there are many different reduced-round variants worth considering. Here, we focus on two: A variant with the full "wrapper" but fewer rounds of cryptographic core, and a variant with both the core and wrapper reduced by the same number of rounds. In other work [KS00], we have considered the cryptographic core without the wrapper. In view of the stated purpose of the "core" and "wrapper" rounds, we believe the two variants in this paper have a great deal to teach us about the ultimate strength of MARS.

### 1.1 Current Results

Our results are as follows:

**Attacks on Reduced-Round MARS Variants**

| Reduced-Round Version | Work | Memory | Text Requirements |
|---|---|---|---|
| Full Mixing + 5 Core | $2^{232}$ half encs | $2^{236}$ bytes | 8 known plain |
| Full Mixing + 5 Core | $2^{247}$ partial encs | $2^{197}$ bytes | $2^{50}$ known plain |
| 6 Mixing + 6 Core | $2^{197}$ partial decs | $2^{73}$ bytes | $2^{69}$ chosen plain |
| 0 Mixing + 11 Core | $2^{229}$ partial encs | $2^{69}$ bytes | $2^{65}$ chosen plain |

For reasons of both space and clarity of presentation, the attacks against the core rounds only are not included in this paper, and can be found in [KS00].

### 1.2 Implications of the Results

None of our current results on MARS come close to breaking the full cipher; our best results to date break only 21 out of 32 rounds (and this counts attacking 16 mixing rounds, which are far weaker than the core rounds). However, the attacks in this paper demonstrate ways to attack the MARS structure, and so highlight potential weaknesses of that structure. They also help us to understand how the components of this complex cipher interact to resist attack.

We also introduce a new kind of meet-in-the-middle attack, which may be of independent interest. Although we demonstrate its use initially on MARS, it may be useful against other ciphers, especially other ciphers with heterogenous structures.

### 1.3 Guide to the Rest of the Paper

The remainder of the paper is arranged as follows: First, we discuss the structure of MARS, and introduce notation and terminology to describe its inner workings. Next, we discuss a set of attacks on MARS with only the number of core rounds reduced. Next, we develop attacks on MARS variants with the same number of each kind of round taken out. We conclude the paper with a discussion of the new techniques we have developed for MARS, the implications of our results, and some open questions.

## 2 The MARS Structure

The MARS structure can be considered as six different layers through which a plaintext block must pass to become a ciphertext block:

1. Pre-Whitening Layer: The plaintext has 128 bits of key material added to its words modulo $2^{32}$.
2. Forward Mixing Layer: Eight rounds of unkeyed mixing operations making extensive use of the MARS S-box.
3. Forward Core Layer: Eight rounds of keyed unbalanced Feistel cipher, using a combination of S-box lookups, multiplications, data-dependent rotations, additions, and XORs to resist cryptanalytic attack.
4. Backward Core Layer: Eight rounds of keyed unbalanced Feistel cipher, using a combination of S-box lookups, multiplications, data-dependent rotations, additions, and XORs to resist cryptanalytic attack.
5. Backward Mixing Layer: Eight rounds of unkeyed mixing operations making extensive use of the MARS S-box.
6. Post-Whitening Layer: The block has 128 bits of key material subtracted from its words modulo $2^{32}$.

In this paper, we typically discuss MARS in terms of two different components. The forward and backward core layers together make up the "cryptographic core"; this core looks like a relatively conventional block cipher, and appears to be reasonably resistant to attack. The pre-whitening, forward mixing layer, backward mixing layer, and post-whitening layer together make up the "wrapper," which protects the cryptographic core from various kinds of attack by requiring a large key guess or some clever cryptanalysis to gain access to inputs and outputs of the core. This is a very different block cipher design than is used in the other AES candidates. Among other things, this new design makes it relatively difficult to determine how to come up with reduced-round variants of the cipher to attack.

### 2.1 The Cryptographic Core

The strength of MARS resides fundamentally in the strength of the core rounds. Both forward and backward core rounds use the same $E$ function, which takes one 32-bit input and two subkey words, and provides three 32-bit words. Each output is combined into one of the three other words. The only difference between forward and backward rounds is the order in which the outputs are combined with the words. The core rounds' strength is based primarily on mixing incompatible operations in the $E$ function, and in their target-heavy Feistel structure, which causes both linear and differential characteristics to quickly spread out into every word. A full description of the MARS core rounds appears in [BCD+98].

The cryptographic core, with a few additional rounds, could stand alone as a cipher; indeed, this would have been a fairly conventional design. Instead, the MARS design team chose to use a smaller number of core rounds,[1] but to surround the core with a "wrapper."

### 2.2 The Wrapper

The key addition/subtraction and mixing layers surround the core rounds, preventing direct access to the core rounds from either the plaintext or the ciphertext side. While the wrapper itself isn't particularly resistant to cryptanalysis, it is quite different in structure than the core, and it is designed to require guessing of key material before an attacker can learn or control either inputs or outputs to the core.

The mixing layers, like the core, have an unbalanced (target-heavy) Feistel structure, but use only S-boxes and mixing of addition and XOR.

We are a little puzzled by the decision to involve only 128 bits of key material on each side of the core. This leaves the possibility of an attacker guessing his way past either half of the wrapper, and thus seeing either input or output, with a guess of only half the maximum key length. A small change to the design would have involved 256 bits of key material on each side, and thus made partial key

[1] Assuming the same level of performance, adding the wrapper requires reducing the number of core rounds.

guessing worthless as a method of bypassing the wrapper. Below, we consider some attacks that simply guess key material to bypass the wrapper entirely. Inclusion of additional key material would apparently have stopped such attacks at very little cost. While we understand the role of the "wrapper" in helping the core resist attacks, we don't understand why it couldn't fulfill this role just as well with another 128 bits of key material being combined in on each side. Such a change to the design would have rendered many of the attacks we describe in this paper impossible, at a low performance cost.

### 2.3 Reduced-Round MARS Variants

In a conventional cipher design, the rounds are all more-or-less the same except for subkeys (and sometimes round constants). There is an obvious way to develop weakened versions of such ciphers: simply reduce the number of rounds. Because of the very different roles of the different kinds of rounds in MARS, however, there are a number of reduced-round MARS variants that can teach us valuable lessons about the ultimate strength or weakness of MARS.

Some reduced-round variants we have considered include:

**Chopping Off the Beginning or End** We evaluate the strength of most ciphers by considering versions with several of the first or last rounds omitted: first the whitening layers and then several rounds of the mixing layers. This isn't a terribly rewarding way to look at MARS, since it omits important parts of the cipher's structure.

**Core Rounds Only** Because most of the cryptographic strength of MARS apparently resides in the core rounds, it is reasonable to consider the strength of these rounds independently. By developing such attacks, we learn how to attack a fundamental component of the cipher, which may be of use in mounting attacks on the full cipher in the future. For space reasons, most of our analysis of the MARS core is described in another paper [KS00].

**Full Cipher with Reduced Core Rounds** An alternative way to evaluate the strength of MARS is to consider the full cipher, but with fewer core rounds. This allows us to see how the core rounds might be attacked, even through the whitening and mixing layers that wrap the core rounds of the cipher. It also gives us insights into how strong the core needs to be to allow MARS to resist cryptanalysis.

**Symmetric Reductions of the Cipher** In the full MARS, there are four different types of rounds, each repeated eight times, for a total of 32 total rounds. It is reasonable to consider symmetric reductions of this; for example, we can consider a MARS variant with only three or four or six of each kind of round. In some sense, this probably provides more information about attacking the full MARS cipher than other kinds of weakened variant, because all the components of the cipher are present.

We believe the last three can teach us many lessons about the ultimate strength of MARS, both in terms of developing tools for attacking the full cipher, and in terms of evaluating how close the best current attacks come to breaking the full MARS.

## 3 Full Mixing with Reduced Core Rounds

In this section, we consider attacks on a MARS variant with the full "wrapper," but a reduced "cryptographic core." These attacks demonstrate how it is possible to mount attacks on a cryptographic core, even through the full wrapper, albeit against a much-weakened core. These attacks penetrate by far the largest number of rounds of the cipher, because they focus on the relatively weak mixing rounds, rather than the much stronger core rounds.

Our attacks in this section are meet-in-the-middle attacks, requiring enormous memory resources to implement, and thus purely academic. In the remainder of this section, we will assume that one memory access to these huge memory devices costs about the same amount of work as a partial encryption. There are ways to trade off time for memory in these attacks, but they generally aren't useful in the context of these attacks.

### 3.1 A Straightforward Meet-in-the-Middle Attack on Five Core Rounds

Consider MARS with full mixing and whitening layers, but with the core reduced to three forward and two backward core rounds. This is vulnerable to a meet-in-the-middle attack as follows:

1. Request eight plaintext/ciphertext pairs.
2. From the plaintext side, guess:
   (a) The 128-bit pre-whitening key.
   (b) The 62-bit first round key.
   (c) $K^{\times}$ and the low nine bits of $K^{+}$ for the second round.
   (d) This yields knowledge of $A_2 = D_3 >>> 13$. Compute this value for all eight plaintexts, and put the resulting 256-bit value in a sorted list.
3. From the ciphertext side, guess:
   (a) The 128-bit post-whitening key.
   (b) The 62-bit last round key.
   (c) $K^{\times}$ and the low nine bits of $K^{+}$ for the next-to-last round.
   (d) This yields knowledge of $A_2 = D_3 >>> 13$. Compute this value for all eight ciphertexts, and search the sorted list from the plaintext guesses for a match on this 256-bit value.

This attack passes through 16 mixing rounds and 5 core rounds (thus 21 rounds total), at a cost of about $2^{232}$ half encryptions' work (that is, $2^{229}$ work for each of the eight texts), and about $2^{236}$ bytes of memory. The memory requirements are totally unreasonable in practice, so this attack is purely academic.

**Summary of Results**

| | |
|---|---|
| **Attack On:** | Full Mixing Plus Five Core Rounds (21 total rounds) |
| **Attack Type:** | Meet-in-the-Middle |
| **Work:** | $2^{232}$ half-encryptions |
| **Memory:** | $2^{236}$ bytes |
| **Texts:** | Eight known plaintexts |

### 3.2 The Differential Meet-in-the-Middle Attack

Here, we introduce the concept of a differential meet-in-the-middle attack. This attack is related to the attack on the Mansour-Even construction by Daemen [Dae95], the attack on DESX by Kilian and Rogaway [KR96], and the inside-out attack of Wagner [Wag99].

In a standard meet-in-the-middle attack, we guess some key from the first and second halves of the cipher, and then match on some middle value. For example, an attack on double-DES starts by getting two plaintexts and their corresponding ciphertexts. We then guess the key for the first DES encryption, and for each such key guess, we compute the middle value from the two plaintexts if they were encrypted under that key. This is stored in a sorted list. We then guess the second DES key, and compute, for each guess, the middle value from decrypting the two ciphertexts. These values are searched for in the sorted list. When we find a matching value, it is very likely that this corresponds to the right key.

This attack can be generalized. For example, it is not necessary that the whole intermediate value to an encryption be computed; we can compute a single bit from each direction, and then examine more plaintext/ciphertext pairs. Similarly, if we can compute some checksum from intermediate values we reach by key guesses from the plaintext and ciphertext sides, then we need never have any knowledge of actual intermediate text values, as in [KSW99].

An extension to this idea allows the use of probability one differentials through some intermediate part of the cipher. Consider the truncated differential $(0, 0, 0, \delta_0) \rightarrow (\delta_1, 0, 0, 0)$, which goes through three MARS core rounds with probability one. The truncated differential works the same way in reverse, naturally. This means that if we see a right input pair (a pair with difference $(0, 0, 0, \delta_0)$), we will also see a right output pair (a pair with difference $(\delta_1, 0, 0, 0)$).

In a meet-in-the-middle attack, we must find some value that can be computed from both the top (input) and the bottom (output) of the cipher with a key guess, build a sorted list of these values, and look for pairs of keys that match on these values.

With these differentials, we can compute such a value as follows:

1. Get about $2^{50}$ known plaintexts and their corresponding ciphertexts. Label each plaintext/ciphertext pair with an index number, $0..2^{50} - 1$.
2. Guess part of the key from the top, and compute intermediate states for each plaintext given that key guess.
3. Sort the plaintext-intermediate values on their first three words.
4. Go through these values, and note each pair of texts that matches on their first three words by their index numbers. List these in sorted order, lower index number first in each pair. We expect about eight of these pairs.
5. Guess part of the key from the bottom, and compute intermediate states for each ciphertext from that key guess. Sort the ciphertext-intermediate values on their last three words.

6. Go through these values, and note each pair of texts that matches on their last three words by their index numbers. List these in sorted order, lower index number first in each pair. We expect about eight of these pairs.
7. Because the differential has probability one in both directions, there must be the same number of these pairs, and the pairs must be identical, from both plaintext and ciphertext. All we've done here is to list which pairs have the right input and output XOR differences to fit this truncated differential.

From this, we now have a "checksum" (the right pair indices) that we can compute across three MARS core rounds. (We can easily restrict the checksum's size to four or eight matching pairs. The indices of the pairs must be put in some standard order; for example, note each right input pair of indices in sorted order, and then sort the pairs in order of each pair's lowest index number.) This checksum costs about $50 \times 2^{50} \approx 2^{56}$ work to find for any block of $2^{50}$ texts. We can thus do the following differential meet-in-the-middle attack on the full MARS mixing layers plus five rounds of core:

1. Get $2^{50}$ known plaintext/ciphertext pairs, and label each by an index number as described above.
2. From the plaintext side, guess the pre-addition key and the first core round key, a total of $2^{190}$ different key guesses.
3. For each key guess, take the predicted inputs to the second core round, and compute the input right pair indices as described above. This takes about $2^{56}$ work per key guess. Write the input right pair indices to a huge list, one entry per key guess with the first eight right input pair indices in sorted order.
4. Do the same thing from the bottom, continuing to add entries to the huge list.
5. Sort the huge list, which will now have $2^{191}$ entries in it, and should thus take about $191 \times 2^{191} \approx 2^{199}$ work to sort.
6. Find the match between key guesses from the plaintext and ciphertext sides.

The total work done is thus $2^{190} \times 2^{56} \times 2 + 2^{199} \approx 2^{247}$. The attack recovers all key material used in the pre- and post-addition/subtraction keys, and the first and last core rounds' values, as well. The total memory taken is $56 \times 2^{191}$ bytes.

**Summary of Results**

| | |
|---|---|
| **Attack On:** | Full Mixing Plus Five Core Rounds (21 total rounds) |
| **Attack Type:** | Differential Meet-in-the-Middle |
| **Work:** | $2^{247}$ partial encryptions |
| **Memory:** | $2^{197}$ bytes |
| **Texts:** | $2^{50}$ known plaintexts |

### 3.3 Tradeoffs Between Differential and Conventional Meet-in-the-Middle Attacks

Note that the differential meet-in-the-middle attack requires slightly more work but considerably less memory than the conventional meet-in-the-middle attack.

The advantage of the differential meet-in-the-middle attack is that it allows us to pass through three core rounds for free; the disadvantage is in the cost of detecting the property that passes through those three core rounds for free, and the far larger number of known plaintexts required. This tradeoff determines which attack is best-suited for a given cipher and attack model.

For reference, we will point out that the differential meet-in-the-middle attacks can be used with less memory against smaller numbers of core rounds. For example, we can use the same truncated differential and filtering process against three rounds of core, dropping the total memory required to about $2^{133}$ bytes of memory, at a work factor of about $2^{185}$ partial encryptions.

We have considered ways of extending the differential meet-in-the-middle attack another round. Unfortunately, there are complications involved in using either differentials with probability substantially lower than one, or in using differentials that don't run both directions with approximately equal probability.

### 3.4 Using Lower Probability Differentials

Consider a differential with probability 1/2 through several rounds of some cipher, and assume we must find four input right pairs that are also output right pairs. The problem is that we must have an exact match for the final sorting and searching phase of the meet-in-the-middle attack to work. The only way we can see to mount the attack in this situation is to generate and store many different input right pairs, in hopes that one will consist of all successful differentials, and thus, right output pairs.

The most efficient way to do this will probably be to find $R$ right input pairs, and add to the sorted list of input right pairs every possible 4-tuple of the pairs, and then to do the same with the right output pairs. That will involve $R$ choose 4 entries in the list, and we can expect it to work if we expect at least 4 of the $R$ input right pairs to result in output right pairs. The number of expected right output pairs from $R$ right input pairs is binomially distributed; for reference, with nine right input pairs, we expect four right output pairs with probability 1/2. With twenty right input pairs, we have about a 0.94 probability of getting some subset of four right input pairs.

This implies an unpleasant tradeoff between probability of the differential used, and memory required for the attack. Consider the following numbers, which describe the impact of using lower-probability differentials on the difficulty of the attack. These numbers are for parameters that give the attack an approximate probability of success of 1/2.

**Memory vs. Probability Tradeoff**

| Prob. of Characteristic | Num. Right Input Pairs Required | Num. Entries in Sorted List per Key Guess |
|---|---|---|
| 0.9 | 5 | 5 |
| 0.5 | 9 | 126 |
| 0.1 | 47 | 178365 |
| 0.01 | 467 | $1.96 \times 10^9$ |
| 0.001 | $\approx 5000$ | $2.60 \times 10^{13}$ |

As a rule, multiplying the number of entries in the sorted list per key guess by $N$ multiplies the size of that list by $N$, which multiplies the work involved in handling it by $N \log N$. We thus have great difficulty in using differentials with very low probabilities.

**Truncated Differentials with Substantially Lower Probabilities in the Decryption Direction** Normal differentials must have the same probability in both directions. (This can be established by a simple counting argument.) However, truncated differentials, which don't specify the whole difference, can have different probabilities in different directions. For example, in the MARS forward core rounds, the following four-round differential has probability one:

$$(0, 0, 0, 2^{31}) \rightarrow (?, ?, (\text{low 12 bits} = 0x1000), 2^{12})$$

However, this truncated differential cannot be run backwards with reasonable probability. There are about $2^{211} + 2^{127}$ pairs of inputs that will yield this output difference; of these pairs, only about $2^{-84}$ have input difference $(0, 0, 0, 2^{31})$. This makes the attack much more costly; in fact, our best methods to mount the attack in this case allow us to attack the full mixing and four rounds of core, but not five rounds of core.

### 3.5 Boomerang Meet-in-the-Middle Attacks

We have also considered using the same kind of technique, but with boomerangs [Wag99] (basically, 4-tuples with a differential relationship between all four texts in the middle of the cipher) instead of individual ciphertexts. The problem of detecting when we have the expected boomerangs is difficult; thus far, we have been unable to find a way to do this that isn't far costlier than the rest of the attack can afford.

### 3.6 Other Techniques

In this section, we have focused on meet-in-the-middle attacks, because these are the most obvious kinds of attacks to consider. However, there are other attacks that might be useful against this kind of reduced-round MARS version. For example, we might guess our way past the pre-addition key and forward mixing layers, and look for a set of text pairs whose properties will show through eight rounds of backward mixing layer. We haven't yet found an effective way to do this for all eight backward mixing rounds, but research is ongoing.

## 4 Symmetric Reductions of the Cipher

MARS consists of eight rounds each of four kinds of round functions. A natural way to derive a reduced-round version of MARS to analyze is to consider $k$ rounds of each kind. For example, when $k = 2$, we have eight total rounds; when

$k = 3$, twelve; and when $k = 4$, sixteen. Cryptanalysis of such reduced-round versions of the cipher allows us to learn important lessons about how to attack a the general MARS structure.

Our attacks typically work as follows:

1. First, we choose $N$ batches of input pairs, so that one such batch is likely to consist of many pairs that have some differential after the mixing layer.
2. We then exploit some differential property that passes through the core rounds, leaving a detectable differential property somewhere near the end of the cipher.
3. Finally, we guess enough key material at the end to detect the detectable property; the partial key guess that allows us to detect the differential property is the correct one.

### 4.1 Attacking MARS Symmetrically Reduced to Eight Rounds

When $k = 2$ (eight rounds total), we have a cipher that is obviously not very strong. It is still worthwhile to consider how this might be attacked, in part to help develop techniques for attacking stronger versions. Recall that this cipher consists of the key addition, the first two forward mixing rounds, two forward core rounds, two backward core rounds, the last two backward mixing rounds, and the key subtraction.

For this version, we can simply use one of the meet-in-the-middle attacks discussed in the previous section, since there are only four rounds. However, we can do much better than that.

Our attack works as follows:

1. We choose $N$ batches of eight pairs each, where $N \leq 2^{40}$. The batches will be described below; one batch will have all eight pairs with difference $(0, 0, 0, 2^{31})$ after the forward mixing layer.
2. In the right batch, this passes through the four core rounds with probability one, leaving $(?, ?, ?, 2^{12})$.
3. We guess a few bits of subtraction key at the end of the cipher, and thus distinguish the right batch from all the wrong batches. If we guess $m$ bits of effective key, we will need about $2^{m+44}$ partial decryptions to distinguish the right batch from the wrong batches.

**Choosing the Batches** The first step to this attack is to get pairs of texts through two forward mixing rounds, so that we have pairs with the difference $(0, 0, 0, 2^{31})$ in the input to the first core round.

Our plan is to put a $2^7$ difference in $A$, and an offsetting difference $T$ in $B$, and finally a difference to cancel $A$'s difference in $D$. To simplify the filtering problem at the end, we will choose *batches* of eight pairs of texts, so that one of the batches will give us eight right pairs through the forward mixing layer.

*Choosing $A, A^*$* We show the first difference as being $2^7$, in $A$, which passes through the key addition with approximate probability $1/2$, and then generates expected difference $T$ in the output to the first use of S-box $s_0$ with probability $2^{-7}$. This then has probability of $2^{-8}$. This is based on simply looking for a pair of S-box inputs, $(u, u \oplus 2^7)$, such that $s_0[u] \oplus s_0[u \oplus 2^7] = T$. We look at all 128 such pairs, and use the difference with the lowest weight in its low 31 bits, for reasons that will become clear momentarily.

For each batch, we hold the low eight bits of $A$ constant. For one such value, $A, A^* = A \oplus 2^7$ will leave a $2^7$ difference in $A$, and a $T$ difference in the output from the first $s_0$.

*Choosing $B, B^*$* The second difference is shown as being $T$, in $B$. This passes through the key addition with approximate probability $2^{-w(T)}$, where $w(T)$ is the Hamming weight of the low 31 bits of $T$. If we get $T$ as the XOR differences in both line $B$ and the $s_0$ output from $A$, they cancel out with probability one. (We can also consider a mod $2^{32}$ difference $T$ that passes through the key addition with probability one, and cancels out the $T$ additive difference in the $s_0$ output with probability about $2^{-w}$; naturally, there is no difference in the probabilities involved.) Recall that we chose $u, u \oplus 2^7$ in $A$ to minimize $w(T)$. Let us assume that the minimum value for $w(T)$ is 12. Then, we have about $2^{-12}$ probability of finding a pair $B, B^*$ such that their difference after the key addition is $T$, simply by using the rule that $B^* = B \oplus T$. We can actually do somewhat better than this in our selection of batches.

*Building the Batches* The third difference is shown as being $2^{31}$, in $D$. This difference passes through all XORs and additions with probability one.

We can thus build batches of $(A, B, C, D), (A^*, B^*, C, D^*)$ pairs. Each batch of eight pairs contains the same low-order eight bits for $A$ and all the same bits for $B$. There are thus $2^{24} \times 2^{32} \times 2^{32} = 2^{88}$ possible pairs for each batch, and there are $2^{40}$ batches possible, and about $2^{12} \times 2^8 = 2^{20}$ expected to be necessary.

We build $2^{20}$ batches of eight pairs, for a total of $2^{24}$ chosen plaintexts.

**Guessing Key at the End** After the core rounds, input pairs with difference $(0, 0, 0, 2^{31})$ must have output difference $(?, ?, ?, 2^{12})$. We must thus learn the value of $D$ in the output from the core rounds. To do this, we must guess about 12 bits of the subtractive key for $C$, and all of the subtractive key for $D$. Using these guesses, we can derive the values for $D$ after the core rounds. We must do this for all $2^{24}$ texts. We thus have total work of about $2^{24} \times 2^{44} = 2^{68}$ partial decryptions. (We suspect that there are better attacks for $k = 2$, but that these attacks don't generalize for larger $k$ values.)

**Summary of Results**

| | |
|---|---|
| **Attack On:** | MARS Symmetrically Reduced to Eight Rounds |
| **Attack Type:** | Differential |
| **Work:** | $2^{68}$ partial decryptions |
| **Memory:** | $2^{29}$ bytes |
| **Texts:** | $2^{25}$ chosen plaintexts |

### 4.2 Extending the Attack to $k = 3$ (Twelve Rounds)

We now consider an attack on MARS symmetrically reduced to 12 rounds. Again, the cipher is obviously not very strong. However, the structure is beginning to add difficulties to the attack. We use a boomerang-amplifier to cover the six core rounds with probability $2^{-96}$ for each pair of pairs with difference $(0, 0, 0, 2^{31})$ into the core rounds. With about $2^{50}$ right pairs into the core rounds' input, we expect to see about four right pairs of pairs. These pairs will then pass through six more rounds, and can be detected by examining the whole output blocks from all the texts.

**Requesting Inputs** We use the same input structure as before, but instead of requesting eight pairs for each batch, we request $2^{50}$ pairs for each batch.

**The Boomerang Amplifier** The boomerang-amplifier attack is introduced in [KS00], and is based on the concept of "boomerangs," as described in [Wag99]. The basic idea of the attack involves a property occurring in pairs of pairs of texts.

For the MARS core, we use the batches of input pairs described above to try to find a batch of $2^{50}$ pairs of texts, all of which will have difference $(0, 0, 0, 2^{31})$ in the input to the first core round. Since there is a probability one differential for the core rounds, $(0, 0, 0, 2^{31}) \rightarrow (2^{31}, 0, 0, 0)$, this means that all $2^{50}$ pairs of the batch will have difference $(2^{31}, 0, 0, 0)$ after three core rounds.

Consider the set of $2^{50}$ of these pairs in the batch. We will refer to these pairs as $((W_0, W_0^*), (W_1, W_1^*), ..., (W_i, W_i^*))$ in input to the core rounds, and as $((X_0, X_0^*), (X_1, X_1^*), ..., (X_i, X_i^*))$ after round three. There are about $2^{99}$ *pairs of pairs*. That is, there are about $2^{99}$ different ways to choose two of these pairs out of this batch and look at them together; for example, $((X_i, X_i^*), (X_j, X_j^*))$. Now, consider the difference $(0, 0, 0, a)$, where $a$ is unknown. For any pair of texts to have such a difference, they must collide in 96 bits; the difference thus is expected to occur in $2^{-96}$ of all random pairs of texts. Thus, if $X_i, X_j$ can be considered as a more-or-less random pair of texts (and they apparently can), then the probability that each $i, j$ pair will have this difference is $2^{-96}$, and since we have $2^{99}$ such pairs, we expect about eight pairs $X_i, X_j$ with this difference. However, we know that $X_i \oplus X_i^* = (2^{31}, 0, 0, 0)$ for all $i$. This lets us algebraically show that when $X_i \oplus X_j = (0, 0, 0, a)$, $X_i^* \oplus X_j^*$ must also equal $(0, 0, 0, a)$. This boomerang structure thus amplifies the effect of the low-probability event, making it detectable, since when this happens we get *two* pairs of texts that follow the truncated differential $(0, 0, 0, a) \rightarrow (b, 0, 0, 0)$ over three rounds.

**Distinguishing the Right Key Guess** To distinguish the right key guess, we examine the result of trial partial decryption of a whole batch of pairs at a time. Let $Y_i, Y_i^*$ be the results of encrypting input pair $W_i, W_i^*$ through the whole cryptographic core in some batch. We build a list of all the $Y_i$ and $Y_i^*$ values. We then sort this list on its low-order 96 bits. Next, we go through the

list, and for each pair $Y_i, Y_j$ or $Y_i, Y_j^*$ that matches in those last 96 bits, the pair $i, j$ is added to a sorted list of pairs that collided. Finally, we *count* the number of times each $i, j$ appears in the list. When we see two or more instances of the same $i, j$ occurring twice, we are extremely likely to have a correct key guess.

Recall that we expect eight pairs $i, j$ such that $X_i \oplus X_j = X_i^* \oplus X_j^* = (0, 0, 0, a)$ These will inevitably lead to eight pairs $i, j$ such that $Y_i \oplus Y_j = (b, 0, 0, 0)$ and $Y_i^* \oplus Y_j^* = (b', 0, 0, 0)$. (The property works just as well if the collision occurs between $X_i$ and $X_j^*$, naturally.)

The probability of any given $i, j$ pair having this property after a random permutation has been applied to it is $2^{-192}$ Since there are $2^{99}$ pairs in each batch, we expect no such $i, j$ pairs. The probability of seeing two such pairs in a batch (that is, among $2^{99}$ potential pairs) is about $2^{-186}$. We will be examining $2^{20}$ different batches, each under $2^{128}$ different keys, so we'll have $2^{148}$ total batches to examine in this way. So with overwhelming probability, there will be only one partial key guess that will give us two or more such $i, j$ pairs.

*Summary of the Attack* The attack on 12 rounds ($k = 3$) makes use of a boomerang amplifier. It requires about $2^{20} \times 2^{48} \times 2 = 2^{69}$ texts, about $2^{25}$ bytes of random-access memory (to hold a batch of texts at a time), and about $2^{73}$ bytes of sequential memory to store all the ciphertexts so we can apply our guesses to them. The attack also requires about $2^{128} \times 2^{69} = 2^{197}$ partial decryptions, each consisting of about one quarter of the cipher.

**Summary of Results**

| | |
|---|---|
| **Attack On:** | MARS Symmetrically Reduced to 12 Rounds |
| **Attack Type:** | Boomerang Amplifier |
| **Work:** | $2^{197}$ partial decryptions |
| **Memory:** | $2^{73}$ bytes |
| **Texts:** | $2^{69}$ chosen plaintexts |

## 5 Conclusions

In this paper, we have developed several new attacks on reduced-round versions of MARS. While none of these attacks is able to break the full cipher, we feel that these results provide valuable insights into the security of MARS. We regard these results as preliminary, and would be unsurprised to see moderate improvements in any of our attacks. However, if major improvements in the results are possible, we expect that they will require new techniques. Below, we describe some ideas for additional attacks on reduced-round MARS variants.

### 5.1 Lessons from the Analysis

The results in this paper show the overwhelming importance of the strength of the MARS cryptographic core; we can attack the full mixing layers with only five core rounds, a total of 21 rounds, but can currently attack no more than 11 core rounds.

Our results also show how the "wrapper" layers protect the core rounds from attacks that require large numbers of chosen plaintexts or chosen ciphertexts.

Finally, our results demonstrate that, when evaluating a fundamentally new cipher design, it is important to be able to innovate—to develop new techniques to attack the cipher, rather than merely reusing the standard differential and linear attacks. Because MARS is such an unconventional block cipher, we needed to develop new attacks to get very far in our analysis.

### 5.2 Why This Is Important

The only way we know of actually determining the strength of a cipher is to try to attack it, including reduced-round versions. Proofs of security have proven unreliable; security arguments based on estimates of the best differential and linear characteristics tell us little about what other attacks may be done; design principles that protect against some attacks sometimes allow new attacks in their place; as in Square, where the use of the MDS matrix made differential attacks extremely difficult, while allowing Knudsen's dedicated attack. The history of cryptography is littered with ciphers whose designers were convinced of their security, but whose attackers were not. Without a solid understanding of the security of each of the AES finalists, NIST and the cryptographic community will likely make a final decision on AES based only on performance.

In this paper, we have done some very preliminary analysis of two versions of MARS with reduced rounds. MARS is a complicated enough design that beginning to analyze it involves a significant investment of time (though even conceptually very simple ciphers seem to have much the same property). We hope to see our work spur others to go beyond the very preliminary results in this paper.

### 5.3 Ideas for Future Attacks

We have spent considerable time trying to get boomerangs to work within meet-in-the-middle attacks. A "boomerang-in-the-middle" attack would go through six rounds for free, and thus would be quite powerful. Similarly, there is a seven-round impossible differential through the core rounds; we are as yet unable to find a way to use either of these ideas in a meet-in-the-middle attack. The underlying problem in the case of the boomerang-in-the-middle attack is that a boomerang 4-tuple can be identified only by considering both input and output simultaneously. We have not been able to find a way around this problem so far. The underlying problem with the impossible differential meet-in-the-middle attack is that we can rule out candidate key guesses only by (again) examining right pairs for both input and output simultaneously. We are still looking for a way around this problem, or for a proof than none exists.

In our differential meet-in-the-middle attacks, we dealt with the mixing layers by simply guessing our way past them. We expect significant improvements to the attacks are possible with more analysis of the mixing layers, particularly in terms of partial guessing of key material. We have spent far more time analyzing

the core rounds than the unkeyed mixing layers, and so this is a good area for further research.

Attacking the symmetrically reduced version of the cipher with $k = 4$ apparently requires a better way of choosing inputs than the input structure we discuss above. We hope to find a better input structure, or a better property to push through all eight core rounds. Previous attempts to attack $k = 4$ have exceeded $2^{256}$ work, usually due to the huge plaintext requirements.

It may also be worthwhile to attack variants of MARS that cut off in the middle (after 12 or 16 rounds total); we have some ideas in this direction.

Finally, in future work, we hope to examine how the MARS key schedule functions with various reduced-round variants.

## 6 Acknowledgements

The "extended Twofish team" met for two week-long cryptanalysis retreats during Fall 1999, once in San Jose and again in San Diego. This paper is a result of those collaborations. Our analysis of MARS and Serpent has very much been a team effort, with everybody commenting on all aspects. The authors would like to thank Niels Ferguson, Mike Stay, David Wagner, and Doug Whiting for useful conversations and comments on these attacks, and for the great time we had together. The authors would also like to thank Susan Langford for helpful suggestions on one of the attacks, and Beth Friedman for proofreading the final paper.

## References

[ABK98] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 98.

[Ada98] C. Adams, "The CAST-256 Encryption Algorithm," NIST AES Proposal, Jun 98.

[BBS99] E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials," *Advances in Cryptology — EUROCRYPT '99 Proceedings*, Springer-Verlag, 1999, pp. 12–23.

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS — A Candidate Cipher for AES," NIST AES Proposal, Jun 98.

[Bih94] E. Biham, "New Types of Cryptanalytic Attacks Using Related Keys," *Journal of Cryptology*, v. 7, n. 4, 1994, pp. 229–246.

[Bih95] E. Biham, "On Matsui's Linear Cryptanalysis," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 398–412.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[BW99] A. Biryukov and D. Wagner, "Slide Attacks," *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999.

[Dae95] J. Daemen, "Cipher and Hash Function Design," Ph.D. thesis, Katholieke Universiteit Leuven, Mar 95.

[DR98] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 98.

[HKM95] C. Harpes, G. Kramer, and J. Massey, "A Generalization of Linear Cryptanalysis and the Applicability of Matsui's Piling-up Lemma," *Advances in Cryptology — EUROCRYPT '95 Proceedings*, Springer-Verlag, 1995, pp. 24–38.

[HKR+98] C. Hall, J. Kelsey, V. Rijmen, B. Schneier, and D. Wagner, "Cryptanalysis of SPEED," *Selected Areas in Cryptography*, Springer-Verlag, 1998, to appear.

[HM97] C. Harpes and J. Massey, "Partitioning Cryptanalysis," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 13–27.

[KM96] L.R. Knudsen and W. Meier, "Improved Differential Attacks on RC5," *Advances in Cryptology — CRYPTO '96*, Springer-Verlag, 1996, pp. 216–228.

[Knu94a] L.R. Knudsen, "Block Ciphers — Analysis, Design, Applications," Ph.D. dissertation, Aarhus University, Nov 1994.

[Knu95b] L.R. Knudsen, "Truncated and Higher Order Differentials," *Fast Software Encryption, 2nd International Workshop Proceedings*, Springer-Verlag, 1995, pp. 196–211.

[KR96] J. Kilian and P. Rogaway, "How to Protect DES Against Exhaustive Key Search," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 252–267.

[KRR+98] L.R. Knudsen, V. Rijmen, R. Rivest, and M. Robshaw, "On the Design and Security of RC2," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 206–221.

[KRW99] L.R. Knudsen, M.B.J. Robshaw, and D. Wagner, "Truncated Differentials in Skipjack," *Advances in Cryptology — CRYPTO '99 Proceedings*, Springer-Verlag, 1999, 99. 165–180.

[KS00] J. Kelsey and B. Schneier, "Initial Cryptanalysis of the MARS Core," draft.

[KSW99] J. Kelsey, B. Schneier, and D. Wagner, "Key Schedule Weaknesses in SAFER+," *The Second Advanced Encryption Standard Candidate Conference*, 1999, pp. 155–167.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91 Proceedings*, Springer-Verlag, 1991, pp. 17–38.

[LH94] S. Langford and M. Hellman, "Differential-Linear Cryptanalysis," *Advances in Cryptology — CRYPTO '94*, Springer-Verlag, 1994.

[Mat94] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 386–397.

[NIST97a] National Institute of Standards and Technology, "Announcing Development of a Federal Information Standard for Advanced Encryption Standard," *Federal Register*, v. 62, n. 1, 2 Jan 1997, pp. 93–94.

[NIST97b] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," *Federal Register*, v. 62, n. 117, 12 Sep 1997, pp. 48051–48058.

[NSA98] NSA, "Skipjack and KEA Algorithm Specifications," Version 2.0, National Security Agency, 29 May 1998.

[RRS+98] R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 98.

[Saa99] M. Saarinen, "A Note Regarding the Hash Function Use of MARS and RC6," available online from `http://www.jyu.fi/ mjos/`, 1999.

[SK96] B. Schneier and J. Kelsey, "Unbalanced Feistel Networks and Block Cipher Design," *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 121–144.

[SKW+98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, Jun 98.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

[Vau96] S. Vaudenay, "An Experiment on DES Statistical Cryptanalysis," *3rd ACM Conference on Computer and Communications Security*, ACM Press, 1996, pp. 139–147.

[Wag99] D. Wagner, "The Boomerang Attack," *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999, pp. 156–170.

# Impossible Differential on 8-Round MARS' Core

Eli Biham* Vladimir Furman†

March 15, 2000

**Abstract**

MARS is one of the AES finalists. The up-to-date analysis of MARS includes the discovery of weak keys, and Biham's estimation that a 12-round variant of MARS is breakable. This estimation was partly founded based on a 7-round impossible differential of the core of MARS. However, no such attack was presented to-date. In this paper we present two new longer impossible differentials of 8 rounds.

## 1 Introduction

MARS[5] is a block cipher designed by IBM as a candidate for the Advanced Encryption Standard selection process, and was accepted as one of the five finalists.

The up-to-date analysis of MARS includes weak keys, and Biham's estimation that MARS reduced to 12 rounds can be attacked[2]. This estimate was partially based on the existence of a 7-round impossible differential of MARS[1] (see [3, 4, 6] for more details on attacks using impossible differential ). In this paper we introduce two 8-round impossible differentials of MARS' core.

## 2 An 8-Round Impossible Differential

We denote binary numbers with a subscript $b$, and a 32-bit binary numbers whose all bits except of bit $i$ are all zero, and only bit $i$ is one by $\delta_i = 0^{31-i}1^1 0^i_b$ (i.e., `1<<i` in C). We also denote a string of 0's (and 1's) of variable lengths (including zero length) by $0^*_b$ (and $1^*_b$) and the complement of a bit-value $x$ by $\bar{x}$ ($\bar{x} = 1 - x$).

*Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel. biham@cs.technion.ac.il, http://www.cs.technion.ac.il/~biham/.

†Computer Science Department, Technion - Israel Institute of Technology, Haifa 32000, Israel. vfurman@cs.technion.ac.il.

The 7-round impossible wordwise (truncated) differential of MARS is of the form

$$(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0) \overset{1\ round}{\not\mapsto} (0,0,0,W) \overset{3\ rounds}{\rightarrow} (Z,0,0,0)$$

where $W$, $X$, $Y$, and $Z$ are non-zero, all pairs with differences of the form $(0,0,0,X)$ must have differences of the form $(Y,0,0,0)$ after 3 rounds, and similarly the differences $(0,0,0,W)$ always cause differences $(Z,0,0,0)$ after 3 rounds. However, there are no pairs with differences $(Y,0,0,0)$ such that the differences become $(0,0,0,W)$ after one round.

We observe that an extension of this impossible differential shows that when $W = \delta_{31}$ the intermediate one-round impossible differential can be replaced by a two-round impossible differential $(Y,0,0,0) \overset{2\ rounds}{\not\mapsto} (0,0,0,\delta_{31})$, for some values of $Y$, leading to the following 8-round impossible differential for some values of $X$

$$(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0) \overset{2\ rounds}{\not\mapsto} (0,0,0,\delta_{31}) \overset{3\ rounds}{\rightarrow} (\delta_{31},0,0,0).$$

In the following we describe the 3-round differentials with probability 1. Then, we describe why the 2-round intermediate differential is impossible, and for which values of $Y$. The conjunction of the various differentials to the 8-round impossible differentials is described at the end of this section.

## 2.1 The 3-Round Differentials with Probability 1

We denote additive difference by $\Delta$, and XOR-differences by $\Delta_{xor}$. In every round of MARS' core, every single 32-bit input word $B$, $C$ and $D$ influences only one 32-bit output word (on $A$, $B$ and $C$ respectively). Thus if we take the input difference of one of the foregoing to be non-zero (e.g., $\Delta B \neq 0$) and all others including $\Delta A$ to be 0 (e.g., $\Delta A = \Delta C = \Delta D = 0$), then we receive the output difference with only one non-zero difference. In particular, if we take some input difference $(0,0,0,X)$ where $X$ is non-zero, we get the difference $(0,0,X_1,0)$ for some non-zero $X_1$ after one round, then the difference becomes $(0,X_2,0,0)$ for some non-zero $X_2$ after the next round. Finally, the difference becomes $(Y,0,0,0)$ for some non-zero $Y$ after the third round. In total we get a 3-round truncated differential $(0,0,0,X) \rightarrow (Y,0,0,0)$ with probability 1.

Note that, if the least significant bits of $X$ have the form $1\underbrace{0..0}_{i}$ $(i \geq 0)$, then the least significant bits of $Y$ have the same form. It follows from the fact that the least significant bits of such form are preserved in both additive and XOR differences.

In the particular case $X = \delta_{31}$ we always get $Y = \delta_{31}$: We start with the following difference $(0,0,0,\delta_{31})$, i.e., $\Delta A_0 = \Delta B_0 = \Delta C_0 = 0, \Delta D_0 = \delta_{31}$. Since $\Delta A_0 = 0$, the mixings to $B$, $C$, and $D$ have zero differences. Since the difference

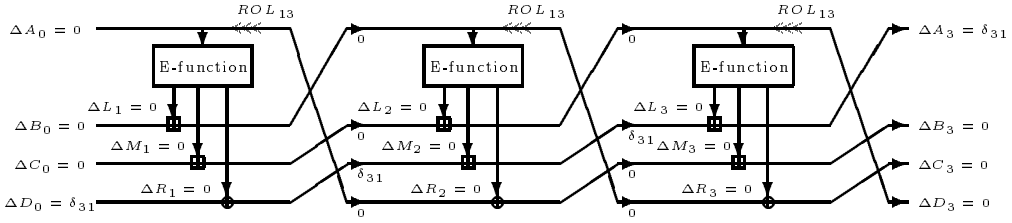



Figure 1: 3-round differential

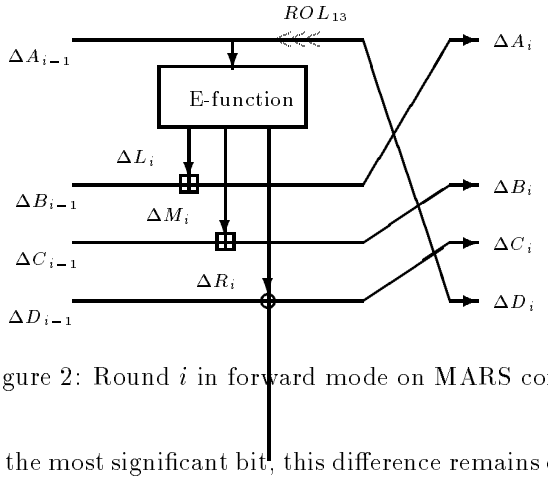



Figure 2: Round $i$ in forward mode on MARS core

in $\Delta D$ is only in the most significant bit, this difference remains only in the most significant bit independently of whether the mixing operation is performed by addition or by XOR. Therefore, we get the difference $(\Delta A_1, \Delta B_1, \Delta C_1, \Delta D_1) = (0, 0, \delta_{31}, 0)$ after one round with probability 1. This can be repeated three times, and we get the difference $(\delta_{31}, 0, 0, 0)$ with probability one after 3 rounds, as shown in Figure 1. Notice, that this differential holds in all the rounds of the core including the forward mode, the backward mode and even on the boundary of both.

## 2.2 The 2-Round Impossible Differential

In this section we describe the 2-round impossible differential of MARS core.

Let $(\Delta A_0, \Delta B_0, \Delta C_0, \Delta D_0) = (Y, 0, 0, 0)$, where $Y$ is an unknown value and $(\Delta A_2, \Delta B_2, \Delta C_2, \Delta D_2) = (0, 0, 0, \delta_{31})$. We want to find the values of $Y$ that give impossible differential on a 2-round MARS core. We look for these values separately in the cases of forward and backward modes.

### 2.2.1 Forward Mode

Figure 2 outlines one round of the forward mode.

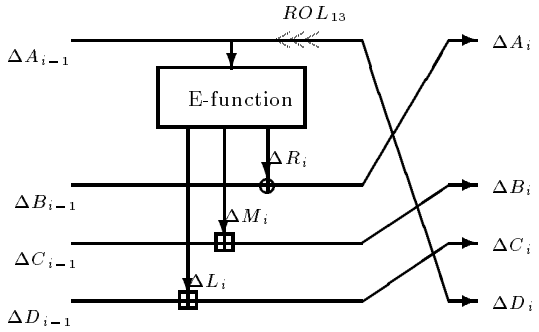



Figure 3: Round $i$ backward mode on MARS core

- We know that $R_i = ((A_{i-1} <<< 13) \cdot K) <<< 10 = (D_i \cdot K) <<< 10$, where $K$ is an unknown subkey. Because, the key used in this stage is odd and $\Delta D_2 = \delta_{31}$, we have that $\Delta_{xor} R_2 = \delta_9$.
- We have $\Delta_{xor} R_2 = \delta_9$ and $\Delta C_2 = \Delta_{xor} C_2 = 0$, so $\Delta_{xor} D_1 = \delta_9$. Thus, we receive $\Delta_{xor} A_0 = \delta_{28}$.
- $\Delta_{xor} A_0 = \delta_{28} \Rightarrow \Delta A_0 = aaa1\underbrace{0..0}_{28}{}_b$, where $a$ is either 0 or 1 (i.e., $\Delta A_0 = \pm\delta_{28}$).

In total, we get that all values of $Y$, with possible exception of $\pm\delta_{28}$, give impossible differentials on a 2-round MARS core in the forward mode.

### 2.2.2 Backward Mode

The Figure 3 outlines the backward mode round.

- $\Delta D_2 = \delta_{31} \Rightarrow \Delta_{xor} D_2 = \delta_{31} \Rightarrow \Delta_{xor} A_1 = \delta_{18}$.
- $\Delta B_0 = 0 \Rightarrow \Delta_{xor} B_0 = 0$; Together with $\Delta_{xor} A_1 = \delta_{18}$ we get that $\Delta_{xor} R_1 = \delta_{18}$.
- $R_i = ((A_{i-1} <<< 13) \cdot K) <<< 10 = (D_i \cdot K) <<< 10$, so $\Delta_{xor}(D_i \cdot K) = \Delta_{xor} R_i >>> 10$. So $\Delta_{xor}(D_1 \cdot K) = \delta_{18} >>> 10 = \delta_8$, and $\Delta(D_1 \cdot K) = \Delta D_1 \cdot K = \pm\delta_8$. Because, the key used in this stage is odd, we have two important conclusions:
  1. $\Delta D_1$ has $\underbrace{10..0}_{9}{}_b$ as a 9 least significant bits.
  2. We may look at this as $(\Delta D_1/2^8) \cdot (K \bmod 2^{24}) = \pm 1$. So the 24 least significant bits of the key are equal to the inverse of $\pm(\Delta D_1/2^8) \bmod 2^{24}$.

- On the other hand:

  $L_2 = (S[9 \text{ least significant bits of } (A_1 + K^+)] \oplus (R_2 >>> 5) \oplus R_2) <<< (5 \text{ least significant bits of } R_2)$,

  where $K^+$ is an unknown subkey.

  - $\Delta_{xor} A_1 = \delta_{18}$ so the 9 least significant bits of $\Delta A_1$ are 0, then $\Delta(9$ least significant bits of $(A_1 + K^+)) = 0$, so $\Delta S = 0$ and thus $\Delta_{xor} S = 0$.
  - As in forward mode, we get $\Delta_{xor} R_2 = \delta_9$, so $\Delta_{xor}(R_2 >>> 5) = \delta_4$.
  - $\Delta_{xor}(S \oplus (R_2 >>> 5) \oplus R_2) = \underbrace{0..0}_{22} 1000010000_b$.
  - A variable rotation is performed on $L_2$ by a number of bits derived from the 5 least significant bits of $R_2$. Because $\Delta_{xor} R_2 = \delta_9$ both rotations are by the same number of bits (denoted by $r_l$), so we have:

    $$\Delta_{xor} L_2 = \underbrace{0..0}_{22} 1000010000_b <<< r_l.$$

  - After the rotation, the result is always of the form:

    $$\Delta_{xor} L_2 = \underbrace{0..0}_{30-i-j}\ 1\ \underbrace{0..0}_{j}\ 1\ \underbrace{0..0}_{i}{}_b,$$

    where $j = 4$ or 26, and $i = 0..30 - j$.
  - Thus we have $\Delta L_2 = \underbrace{b..b}_{30-i-j}\ \bar{a}\ \underbrace{a..a}_{j}\ 1\ \underbrace{0..0}_{i}{}_b$, where a,b are unknown bit values.

- Because $\Delta L_2 + \Delta D_1 = \Delta C_2 = 0$, we have that $\Delta D_1 = \underbrace{\bar{b}..\bar{b}}_{30-i-j}\ a\ \underbrace{\bar{a}..\bar{a}}_{j}\ 1\ \underbrace{0..0}_{i}{}_b$. But we know that $\Delta D_1$ has $\underbrace{10..0}_{9}{}_b$ as the 9 least significant bits, so only a single possibility remains:

  $$\Delta D_1 = \underbrace{\bar{b}..\bar{b}}_{18}\ a\ \underbrace{\bar{a}..\bar{a}}_{4}\ 1\ \underbrace{0..0}_{8}{}_b.$$

  **Observation:** $\Delta D_1$ may have four possible values:

  - $\underbrace{0..0}_{18} 1 \underbrace{0..0}_{4} 1 \underbrace{0..0}_{8}{}_b$ and $\underbrace{1..1}_{18} 0 \underbrace{1..1}_{4} 1 \underbrace{0..0}_{8}{}_b$ (i.e., $\pm \underbrace{0..0}_{18} 1 \underbrace{0..0}_{4} 1 \underbrace{0..0}_{8}{}_b$).
  - $\underbrace{0..0}_{18} 0 \underbrace{1..1}_{4} 1 \underbrace{0..0}_{8}{}_b$ and $\underbrace{1..1}_{18} 1 \underbrace{0..0}_{4} 1 \underbrace{0..0}_{8}{}_b$ (i.e., $\pm \underbrace{0..0}_{19} \underbrace{1..1}_{4} 1 \underbrace{0..0}_{8}{}_b$).

We have two pairs of possible values for $\Delta D_1$, and thus there are only two possible values (one for each pair) for the 24 least significant bits of the key used in first round for multiplication (according to the conclusion in the beginning of this section(2.2.2)). These key values in hexadecimal form are $f07c1f_x$ (for $\Delta D_1 = \pm \underbrace{0..0}_{18}\,1\,\underbrace{0..0}_{4}\,1\,\underbrace{0..0}_{8}{}_b$) and $ef7bdf_x$ (for $\Delta D_1 = \pm \underbrace{0..0}_{19}\,\underbrace{1..1}_{4}\,1\,\underbrace{0..0}_{8}{}_b$).

- It is known that sequences of the form $01^*1_b$ or of the form $10^*1_b$ in the additive difference ($\Delta$) are translated to the sequence of the form either $100^*1^*1_b$ or $01^*1_b$ in the corresponding XOR difference ($\Delta_{xor}$)[1]. Thus we have two options:
  1. $\Delta_{xor} D_1 = \underbrace{0^*1^*}_{18}\underbrace{100^*1^*}_{5}\,1\,\underbrace{0..0}_{8}{}_b$
  2. $\Delta_{xor} D_1 = \underbrace{0^*1^*}_{18}\underbrace{01..1}_{5}\,1\,\underbrace{0..0}_{8}{}_b$
- $\Delta_{xor} A_0 = \Delta_{xor} D_1 >>> 13$, so there are two possible values for $\Delta_{xor} A_0$:
  1. $\Delta_{xor} A_0 = \underbrace{00^*1^*}_{4}\,1\,\underbrace{0..0}_{8}\,\underbrace{0^*1^*}_{18}1_b$
  2. $\Delta_{xor} A_0 = \underbrace{1..1}_{4}\,1\,\underbrace{0..0}_{8}\,\underbrace{0^*1^*}_{18}0_b$
- In the first case, $\Delta_{xor} A_0$ is odd, so the $\Delta A_0$ is odd too, and we cannot show that this case is impossible. In the second case, $\Delta_{xor} A_0$ is even so the $\Delta A_0$ is even too, and therefore we can divide this case in two sub-cases:
  1. There is at least one 1 in $\underbrace{0^*1^*}_{18}{}_b$, so we have $10_b$ as two least significant bits in $\Delta_{xor} A_0$ and $\Delta A_0$. This sub-case is impossible (see Appendix A for a detailed proof).

[1] For checking this fact, look at different cases of such sequence with and without carry from previous bits. For example, we take $\Delta I = 10..01_b$, i.e., $I^1 - I^2 = 10..01_b$. Then either:

1. The least significant bit of $I^1$ is 1: then the least significant bit of $I^2$ must be 0, and thus there is no carry to the next bit. On the other hand, the next bit in the difference is 0. Combining these together we conclude that the next bit in $I^1$ and the next bit in $I^2$ must be equal. Continuing in this way we get that $\Delta_{xor} I = 10..01_b$.
2. The least significant bit of $I^1$ is 0: then the least significant bit of $I^2$ must be 1, and thus there is a carry to the next bit. On the other hand, the next bit in the difference is 0. Combining these together we conclude that the next bit in $I^1$ and the next bit in $I^2$ have different values. Continuing in this way we get that the corresponding bits in $I^1$ and in $I^2$ are different till either: 1) in some bit $I^1$ has 1 and in $I^2$ has 0, or 2) we reach the most significant bits with difference 1 and, due to existence of a carry from the previous bits, this bit in $I^1$ and $I^2$ must have the same value. So $\Delta_{xor} I$ is equal either to $100^*1^*11_b$ or to $01..11_b$.

2. There are no 1's in $\underbrace{0^*1^*}_{18}{}_b$, so $\Delta_{xor}A_0 = \underbrace{1..1}_{4}\,1\,\underbrace{0..0}_{27}{}_b$, and $\Delta A_0$ has $1\,\underbrace{0..0}_{27}{}_b$ as 28 least significant bits. For this sub-case, we cannot show that it is impossible.

Thus, we have a 2-round impossible differential for any **even** $Y$ whose 28 least significant bits are not $\underbrace{10..0}_{28}{}_b$. For other $Y$'s we cannot say anything whether there exist impossible differentials. However, if the differentials are not impossible for some $Y$, then the 24 least significant bits of the multiplication key used in the first round of the differential are either $f07c1f_x$ or $ef7bdf_x$.

### 2.3 Conjunction to the 8-Round Impossible Differentials

We want now to check what values of $X$ give the 8-round impossible differentials. We describe the two cases in which the two middle rounds work in forward mode and in backward mode.

For forward mode, we have a 2-round impossible differential for any value of $Y$, except of $\pm\delta_{28}$. Because in $(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0)$ the relation between $X$ to $Y$ passes through two additions and one exclusive-or operation, the 29 rightmost bits remains $1\,\underbrace{0..0}_{28}{}_b$ and the 3 most significant bits may get any value. So, we have the 8-round impossible differentials $(0,0,0,X) \overset{8\ rounds}{\not\mapsto} (\delta_{31},0,0,0)$ for all $X$, except of those with $\underbrace{10..0}_{29}{}_b$ as the 29 least significant bits.

For backward mode, we have a 2-round impossible differential for any even $Y$, except of those with $\underbrace{10..0}_{28}{}_b$ as 28 least significant bits. As in forward mode, in $(0,0,0,X) \overset{3\ rounds}{\rightarrow} (Y,0,0,0)$ the 28 least significant bits remains $\underbrace{10..0}_{28}{}_b$ and the 4 most significant bits may get any value. So we have the 8-round impossible differentials $(0,0,0,X) \overset{8\ rounds}{\not\mapsto} (\delta_{31},0,0,0)$ for any even $X$, except of those with $\underbrace{10..0}_{28}{}_b$ as the 28 least significant bits.

## 3 Another 8-Round Impossible Differential

There is another 8-round impossible differential on MARS' core:

$$(0,0,0,\delta_{31}) \overset{3\ rounds}{\rightarrow} (\delta_{31},0,0,0) \overset{3\ rounds}{\not\mapsto} (0,0,X,\delta_{31}) \overset{2\ rounds}{\rightarrow} (Y,\delta_{31},0,0),$$

where the 3 middle round are in backward mode, and $X$,$Y$ are non-zero values such that $X$ must have $\underbrace{0..0}_{24}{}_b$ as the least significant bits, and the 8 most significant bits of $X$ may have any value (except of all zeroes). Thus, as was shown in the previous section, $Y$ must have $\underbrace{0..0}_{24}{}_b$ as the least significant bits, and the 8 most significant bits may have any value (except of all zeroes). The explanation for this differential is similar to the explanation described earlier.

## Acknowledgments

We would like to acknowledge the work of Alon Becker and Eran Richardson who made the initial observations in this direction.

## References

[1] A. Becker, E. Richardson, Course Project, 1998.

[2] E. Biham, *A note on Comparing the AES Candidates*, Second AES Conference, March 1999.

[3] E. Biham, A. Biryukov, A. Shamir, *Cryptanalysis of Skipjack Reduced o 31 Rounds Using Impossible Differentials*, LNCS, Advanced in Cryptology - Proceeding of EUROCRYPT'99, Springer-Verlag 1999.

[4] E. Biham, A. Biryukov, A. Shamir, *Miss in the Middle Attacks on IDEA, Khufu, and Khafre*, LNCS 1636, Fast Software Encryption, pp. 124-138, March 1999.

[5] C. Burwick, D. Coppersmith, E. C'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, *"MARS - A Candidate Cipher for AES "*, NIST AES Proposal, June 1998.

[6] L. Knudsen, *DEAL - A 128-bit Block Cipher*, NIST AES Proposal, June 1998.

# A Impossible differential for $Y$, with $10_b$ as least significant bits, in backward mode on MARS core.

In this appendix we show that the sub-case of backward mode where $\Delta_{xor} A_0 = \underbrace{1..1}_{4}\,1\,\underbrace{0..0}_{8}\,\underbrace{0^*1^*}_{17}\,10_b$, mentioned in section 2.2.2, is impossible.

- As in forward mode $\Delta D_2 = \delta_{31} \Rightarrow \Delta_{xor} R_2 = \delta_9$.
- $\Delta_{xor} R_2 \oplus \Delta_{xor} B_1 = \Delta_{xor} A_2 = 0$, so $\Delta_{xor} B_1 = \Delta_{xor} R_2 = \delta_9$.
- $\Delta_{xor} B_1 = \delta_9 \Rightarrow \Delta B_1 = \underbrace{a..a}_{22}\,1\,\underbrace{0..0}_{9}{}_b$, where $a$ is unknown bit value.
- $\Delta C_0 + \Delta M_1 = \Delta B_1 = \underbrace{a..a}_{22}\,1\,\underbrace{0..0}_{9}{}_b$. Because $\Delta C_0 = 0$, $\Delta M_1 = \underbrace{a..a}_{22}\,1\,\underbrace{0..0}_{9}{}_b$.
- $\Delta M_1 = \underbrace{a..a}_{22}\,1\,\underbrace{0..0}_{9}{}_b \Rightarrow \Delta_{xor} M_1 = \underbrace{0^*1^*}_{22}\,1\,\underbrace{0..0}_{9}{}_b$.
- We know that $M_i = (A_{i-1} + K) <<< (\text{low 5 bits of } (R_i >>> 5))$. However, because $\Delta_{xor} R_1 = \delta_{18}$, both rotations are by the same number of bits (denoted $r_m$), and because $\Delta K = 0$ we have
$$\Delta M_1 = \Delta A_0 <<< r_m$$
or
$$\Delta A_0 = \Delta M_1 >>> r_m.$$
- We know that $\Delta_{xor} A_0 = \underbrace{1..1}_{4}\,1\,\underbrace{0..0}_{8}\,\underbrace{0^*1^*}_{17}\,10_b$. It gives us that $\Delta A_0 = \underbrace{x}_{4}\,\bar{a}\,\underbrace{a..a}_{9}\,\underbrace{z}_{17}\,10_b$, where $x, z$ are unknown binary word and $a$ is unknown bit value.
- The $\Delta A_0$ has $10_b$ as 2 least significant bits, so the only one possibility for $r_m$ to be 8. Thus $\Delta_{xor}(M_1 >>> 8) = \underbrace{0..0}_{8}\,\underbrace{0^*1^*}_{22}\,10_b$, and therefore, $\Delta(M_1 >>> 8) = \underbrace{b..b}_{8}\,\underbrace{y}_{22}\,10_b$, where b is an unknown bit value and $y$ is unknown binary word.
- Now we have $\Delta(M_1 >>> 8) = \underbrace{b..b}_{8}\,\underbrace{y}_{22}\,10_b$ and $\Delta A_0 = \underbrace{x}_{4}\,\bar{a}\,\underbrace{a..a}_{9}\,\underbrace{z}_{17}\,10_b$. These must be equal. However, the bit 26th of the later differ than bit 27th, while bits 26th and 27th of the former are equal. This contradicts the fact that both values must be equal.

# Preliminary Cryptanalysis of Reduced-Round Serpent

Tadayoshi Kohno[1*], John Kelsey[2], and Bruce Schneier[2]

[1] Reliable Software Technologies
kohno@rstcorp.com
[2] Counterpane Internet Security, Inc.
{kelsey,schneier}@counterpane.com

**Abstract.** Serpent is a 32-round AES block cipher finalist. In this paper we present several attacks on reduced-round variants of Serpent that require less work than exhaustive search. We attack six-round 256-bit Serpent using the meet-in-the-middle technique, 512 known plaintexts, $2^{246}$ bytes of memory, and approximately $2^{247}$ trial encryptions. For all key sizes, we attack six-round Serpent using standard differential cryptanalysis, $2^{83}$ chosen plaintexts, $2^{40}$ bytes of memory, and $2^{90}$ trial encryptions. We present boomerang and amplified boomerang attacks on seven- and eight-round Serpent, and show how to break nine-round 256-bit Serpent using the amplified boomerang technique, $2^{110}$ chosen plaintexts, $2^{212}$ bytes of memory, and approximately $2^{252}$ trial encryptions.

## 1 Introduction

Serpent is an AES-candidate block cipher invented by Ross Anderson, Eli Biham, and Lars Knudsen [ABK98], and selected by NIST as an AES finalist. It is a 32-round SP-network with key lengths of 128 bits, 192 bits, and 256 bits. Serpent makes clever use of the bitslice technique to make it efficient in software. However, because of its conservative design and 32 rounds, Serpent is still three times slower than the fastest AES candidates [SKW+99].

In the Serpent submission document [ABK98], the authors give upper bounds for the best differential characteristics through the cipher. However, no specific attacks on reduced-round versions of the cipher are presented. In this paper we consider four kinds of attacks on reduced-round variants of Serpent: differential [BS93], boomerang [Wag99], amplified boomerang [KKS00], and meet-in-the-middle. To the best of our knowledge, these are the best published attacks against reduced-round versions of Serpent.[1]

The current results on Serpent are as follows (see Table 1):

1. A meet-in-the-middle attack on Serpent reduced to six rounds, requiring 512 known plaintext/ciphertext pairs, $2^{246}$ bytes of random-access memory, and work equivalent to approximately $2^{247}$ six-round Serpent encryptions.

* Part of this work was done while working for Counterpane Internet Security, Inc.
[1] Dunkelman cryptanalyzed a Serpent variant with a modified linear transformation in [Dun99].

| Rounds | Key Size | Complexity | | | Comments |
|---|---|---|---|---|---|
| | | [Data] | [Work] | [Space] | |
| — | — | — | — | — | no previous results |
| 6 | 256 | 512 KP | $2^{247}$ | $2^{246}$ | meet-in-the-middle (§6) |
| 6 | all | $2^{83}$ CP | $2^{90}$ | $2^{40}$ | differential (§3.2) |
| 6 | all | $2^{71}$ CP | $2^{103}$ | $2^{75}$ | differential (§3.3) |
| 6 | 192 & 256 | $2^{41}$ CP | $2^{163}$ | $2^{45}$ | differential (§3.4) |
| 7 | 256 | $2^{122}$ CP | $2^{248}$ | $2^{126}$ | differential (§3.5) |
| 8 | 192 & 256 | $2^{128}$ CPC | $2^{163}$ | $2^{133}$ | boomerang (§4.2) |
| 8 | 192 & 256 | $2^{110}$ CP | $2^{175}$ | $2^{115}$ | amp. boomerang (§5.3) |
| 9 | 256 | $2^{110}$ CP | $2^{252}$ | $2^{212}$ | amp. boomerang (§5.4) |

KP — known plaintext, CP — chosen plaintext, CPC — chosen plaintext/ciphertext.

**Table 1.** Summary of attacks on Serpent. Work is measured in trial encryptions; space is measured in bytes.

2. A differential attack on Serpent reduced to six rounds, requiring $2^{83}$ chosen plaintexts, $2^{40}$ bytes of sequential-access memory, and work equivalent to approximately $2^{90}$ six-round Serpent encryptions.
3. A differential filtering attack on Serpent reduced to seven rounds, requiring $2^{122}$ chosen plaintexts, $2^{126}$ bytes of sequential-access memory, and work equivalent to approximately $2^{248}$ six-round Serpent encryptions.
4. A boomerang attack on Serpent reduced to eight rounds, requiring all $2^{128}$ plaintext/ciphertext pairs under a given key, $2^{133}$ bytes of random-access memory, and work equivalent to approximately $2^{163}$ eight-round Serpent encryptions.[2]
5. An amplified-boomerang key-recovery attack on Serpent reduced to eight rounds, requiring $2^{110}$ chosen plaintexts, $2^{115}$ bytes of random-access memory, and work equivalent to approximately $2^{175}$ eight-round Serpent encryptions.
6. An amplified-boomerang key-recovery attack on Serpent reduced to nine rounds, requiring $2^{110}$ chosen plaintexts, $2^{212}$ bytes of random-access memory, and work equivalent to approximately $2^{252}$ nine-round Serpent encryptions.

The remainder of this paper is organized as follows: First, we discuss the internals of Serpent and explain the notation we use in this paper. We then use differential, boomerang, and amplified boomerang techniques to break up to nine rounds of Serpent. Subsequently we discuss a six-round meet-in-the-middle attack on Serpent. We then discuss some observations on the Serpent key schedule. We conclude with a discussion of our results and some open questions.

[2] Because this eight-round boomerang attack requires the entire codebook under a single key, one can consider this attack a glorified distinguisher that also recovers the key.

## 2 Description of Serpent

In this document we consider only the bitsliced version of Serpent. The bitsliced and non-bitsliced versions of Serpent are functionally equivalent; the primary difference between the bitsliced and non-bitsliced versions of Serpent are the order in which the bits appear in the intermediate stages of the cipher. Full details of the bitsliced and non-bitsliced version of Serpent are in [ABK98].

### 2.1 The Encryption Process

Serpent is a 32-round block cipher operating on 128-bit blocks. In the bitsliced version of Serpent, one can consider each 128-bit block as the concatenation of four 32-bit words.

Let $B_i$ represent Serpent's intermediate state prior to the $i$th round of encryption. Notice that $B_0 = P$ and $B_{32} = C$, where $P$ and $C$ are the plaintext and ciphertext, respectively.

Let $K_i$ represent the 128-bit $i$th round subkey and let $S_i$ represent the application of the $i$th round S-box. Let $L$ be Serpent's linear transformation. Then the Serpent round function is defined as:

$$\begin{aligned} X_i &\leftarrow B_i \oplus K_i \\ Y_i &\leftarrow S_i(X_i) \\ B_{i+1} &\leftarrow L(Y_i) \qquad i = 0, \ldots, 30 \\ B_{i+1} &\leftarrow Y_i \oplus K_{i+1} \; i = 31 \end{aligned}$$

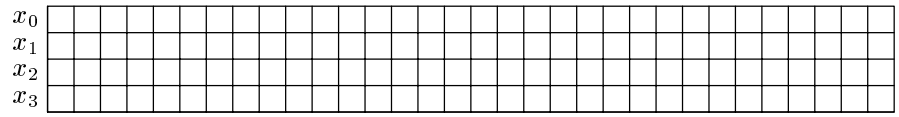
Serpent uses eight S-boxes $S_0, \ldots, S_7$. The indices to $S$ are reduced modulo 8; i.e., $S_0 = S_8 = S_{16} = S_{24}$. The Serpent S-boxes take four input bits and produce four output bits. Consider the application of an S-box $S_i$ to the 128 bit block $X_i$. Serpent first separates $X_i$ into four 32-bit words $x_0$, $x_1$, $x_2$, and $x_3$. For each of the 32-bit positions, Serpent constructs a nibble from the corresponding bit in each of the four words, with the bit from $x_3$ being the most significant bit. Serpent then applies the S-box $S_i$ to the constructed nibble and stores the result in the respective bits of $Y_i = (y_0, y_1, y_2, y_3)$.

The linear transform $L$ on $Y_i = (y_0, y_1, y_2, y_3)$ is defined as

$$\begin{aligned} y_0 &\leftarrow y_0 \lll 13 \\ y_2 &\leftarrow y_2 \lll 3 \\ y_1 &\leftarrow y_0 \oplus y_1 \oplus y_2 \\ y_3 &\leftarrow y_2 \oplus y_3 \oplus (y_0 \ll 3) \\ y_1 &\leftarrow y_1 \lll 1 \\ y_3 &\leftarrow y_3 \lll 7 \\ y_0 &\leftarrow y_0 \oplus y_1 \oplus y_3 \\ y_2 &\leftarrow y_2 \oplus y_3 \oplus (y_1 \ll 7) \\ y_0 &\leftarrow y_0 \lll 5 \\ y_2 &\leftarrow y_2 \lll 22 \\ B_{i+1} &\leftarrow (y_0, y_1, y_2, y_3) \end{aligned}$$

where $\lll$ denotes a left rotation and $\ll$ denotes a left shift.

When discussing the internal state of the Serpent, we will often refer to diagrams such as




where $X_i$ is the internal state under inspection and $X_i = (x_0, x_1, x_2, x_3)$. As suggested by this diagram, we will occasionally refer to an active S-box as a "column."

### 2.2 The Key Schedule

Serpent's key schedule can accept key sizes up to 256 bits. If a 256-bit key is used, Serpent sets the eight 32-bit words $w_{-8}, w_{-7}, \ldots, w_{-1}$ to the key. If not, the key is converted to a 256-bit key by appending a '1' bit followed by a string of '0's.

Serpent computes the prekeys $w_0, w_1, \ldots, w_{131}$ using the recurrence

$$w_i \leftarrow (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11$$

where $\phi$ is `0x9e3779b9`.

Serpent then computes the 128-bit subkeys $K_j$ by applying an S-box to the prekeys $w_{4j}, \ldots, w_{4j+3}$:

$$\begin{aligned}
K_0 &\leftarrow S_3(w_0, w_1, w_2, w_3) \\
K_1 &\leftarrow S_2(w_4, w_5, w_6, w_7) \\
K_2 &\leftarrow S_1(w_8, w_9, w_{10}, w_{11}) \\
K_3 &\leftarrow S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
K_4 &\leftarrow S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\vdots \\
K_{31} &\leftarrow S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
K_{32} &\leftarrow S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}$$

## 3 Differential Cryptanalysis

Differential cryptanalysis, first publicly discussed by Biham and Shamir [BS93], is one of the most well-known and powerful cryptanalytic techniques. Although the original Serpent proposal provided theoretical upper bounds for the highest probability characteristics through reduced-round Serpent variants [ABK98], the Serpent proposal did not present any empirical results describing how successful differential cryptanalysis would be against Serpent in practice.

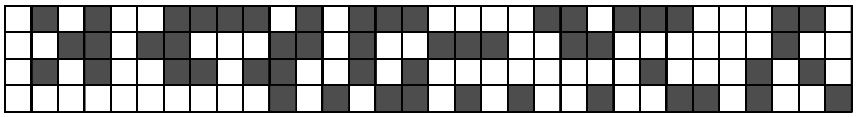
In this section we consider actual differential attacks against reduced-round Serpent variants. Although there may exist other high-probability differentials through several rounds of Serpent, we focus on a particular five-round characteristic, $B'_1 \rightarrow Y'_5$, with probability $p = 2^{-80}$. This characteristic spans Serpent's second through sixth rounds (rounds $i = 1, \ldots, 5$). For completeness, this characteristic is illustrated in Appendix A.1. Notationally, we use $X'$ to represent the XOR difference between two values $X$ and $X^*$.

### 3.1 Basic Six-Round Differential Attack

We can use the above-mentioned five-round, probability $2^{-80}$, characteristic to attack rounds one through six of 192- and 256-bit Serpent.

To sketch our attack: we request $2^{82}$ plaintext pairs with an input difference $B'_1$. For each last round subkey guess, we initialize a count variable to zero. Then, for each unfiltered pair, we peel off the last round and look for our expected output difference from the fifth round. If we observe our expected output difference, we increment our counter. If we count three or more right pairs, we note this subkey as likely to be correct.

If we apply the linear transformation $L$ to the intermediate difference $Y'_5$ (Appendix A.1), we get the following expected input difference to the sixth round:

$B'_6$ 

We can immediately identify all but approximately $2^{-47}$ of our ciphertext pairs as wrong pairs because their differences $B'_7$ cannot correspond to our desired difference $B'_6$.

After filtering we are left with approximately $2^{35}$ ciphertext pairs. Our attack thus requires approximately $2^{36} \times 2^{116}$ partial decryptions, or work equivalent to approximately $2^{150}$ six-round Serpent encryptions. If we retain only our unfiltered ciphertext pairs, this attack requires approximately $2^{40}$ bytes of sequential memory. The signal-to-noise ratio of this attack is $2^{83}$.

### 3.2 Improved Six-Round Differential Attack

By counting on fewer than 116 bits of the last round subkey, we can considerably improve the six-round differential attack in the previous section. For example, if we count on two sets of 56 bits, our work is reduced to about $2^{90}$ Serpent six-round encryptions. This allows us to break six rounds of 128-, 192-, and 256-bit Serpent using less work than exhaustive search.

### 3.3 Bypassing the First Round

We can use structures to bypass the first round of our five-round characteristic $B'_1 \rightarrow Y'_5$. This gives us an attack that requires fewer chosen plaintexts but more work than the attack in Section 3.2. In this attack we use the four-round, probability $2^{-67}$, characteristic $B'_2 \rightarrow Y'_5$. We request $2^{47}$ blocks of $2^{24}$ plaintexts such that each block varies over all possible inputs to the active S-boxes in $B'_1$. This gives us $2^{70}$ pairs with our desired input difference to the second round. We expect eight pairs with our desired difference $Y'_5$.

We can mount the attack in Section 3.2 by looking for the last round subkey suggested seven or more times. In this attack we must consider a total of $2^{94}$ possible ciphertext pairs. As with Section 3.2, we can immediately identify all but $2^{-47}$ of these pairs as wrong pairs. This attack requires work equivalent to approximately $2^{102}$ Serpent six-round encryptions and approximately $2^{75}$ bytes of random-access memory.

### 3.4 Additional Six-Round Differential Attack

We can modify our basic six-round differential attack by guessing part of the last round subkey and looking at the eight passive S-boxes in $B'_5$. In order to do this, we must guess 124 bits of the last round subkey.

In this attack we request $2^{40}$ chosen-plaintext pairs with our input difference $B'_1$. This gives us $2^9$ pairs with difference $B'_5$ entering the fifth round. For a correct 124-bit last round subkey guess, we expect to count $2^9$ pairs with passive S-boxes in $Y'_5$ corresponding to the passive S-boxes in $B'_5$. For an incorrect last round subkey guess, the number of occurrences of pairs with passive differences in our eight target S-boxes is approximately normal with mean $2^8$ and standard deviation $2^4$. Since $2^9$ is 16 standard deviations to the right of $2^8$, we expect no false positives.

This attack requires $2^{45}$ bytes of sequential memory and work equivalent to approximately $2^{163}$ Serpent six-round encryptions.

### 3.5 Seven-Round Differential Filtering Attack

We can use our filtering scheme in Section 3.1 to distinguish six rounds of Serpent from a random permutation. In this distinguishing attack we request $2^{121}$ plaintext pairs with our desired input difference $B'_1$. We expect approximately $2^{41}$ right pairs. Since our filter passes ciphertext pairs with a probability $2^{-47}$, we expect approximately $2^{74} + 2^{41}$ ciphertext pairs to pass our filter.

In a random permutation, the number of unfiltered pairs is approximately a normal distribution with mean $2^{74}$ and standard deviation $2^{37}$. Since $2^{74} + 2^{41}$ is 16 standard deviations to the right of the random distribution's mean of $2^{74}$, we can distinguish six-round Serpent from a random permutation. For a random distribution, the probability of observing $2^{74} + 2^{41}$ or more unfiltered pairs is approximately $2^{-190}$.

We can extend this six-round distinguishing attack to a seven-round key recovery attack on rounds one through seven by guessing the entire last round subkey $K_8$ and performing our six-round distinguishing attack. This attack requires approximately $2^{126}$ bytes of sequential memory $2^{248}$ Serpent seven-round encryptions.

# 4 Boomerang Attacks

## 4.1 Seven-Round Boomerang Distinguisher

In addition to being able to perform traditional differential attacks against Serpent, we can also use Wagner's boomerang attack [Wag99] to distinguish seven rounds of Serpent from a random permutation.

Let us consider a seven-round variant of Serpent corresponding to the second through eighth rounds of the full 32-round Serpent (i.e., rounds $i = 1, \ldots, 7$). Call the first four rounds of this seven-round Serpent $E_0$ and call the final three rounds $E_1$. Our seven-round Serpent is thus $E = E_1 \circ E_0$. We can now apply the boomerang technique to this reduced-round Serpent.

Notice that if we only consider the first four rounds of the five-round characteristic in Appendix A.1, we have a four-round characteristic $B_1' \rightarrow Y_4'$ through $E_0$ with probability $2^{-31}$. Also notice that there exist three-round characteristics through $E_1$ with relatively high probability. Appendix A.2 illustrates one such characteristic, $B_5' \rightarrow Y_7'$, with probability $2^{-16}$.

To use the terminology in [Wag99], let $\Delta = B_1'$, let $\Delta^* = Y_4'$, let $\nabla = Y_7'$ and let $\nabla^* = B_5'$. We then use $\Delta \rightarrow \Delta^*$ as our differential characteristic for $E_0$ and $\nabla \rightarrow \nabla^*$ as our differential characteristic for $E_1^{-1}$.

In the boomerang distinguishing attack, we require approximately $4 \cdot 2^{94}$ adaptive-chosen plaintext/ciphertext queries, or approximately $2^{94}$ quartets $P$, $P'$, $Q$, and $Q'$ and their respective ciphertexts $C$, $C'$, $D$, and $D'$. More specifically, in our distinguishing attack we request the ciphertext $C$ and $C'$ for about $2^{94}$ plaintexts $P$ and $P'$ where $P \oplus P' = \Delta$. From $C$ and $C'$ we compute the ciphertexts $D = C \oplus \nabla$ and $D' = C' \oplus \nabla$. We then apply the inverse cipher to $D$ and $D'$ to obtain $Q$ and $Q'$. For any quartet $P$, $P'$, $Q$, and $Q'$, we expect the combined properties $P \oplus P' = Q \oplus Q' = \Delta$ and $C \oplus D = C' \oplus D' = \nabla$ to hold with probability $2^{-94}$.

## 4.2 Eight-Round Boomerang Key Recovery Attack

We can extend our seven-round boomerang distinguisher to an eight-round key recovery attack on 192- and 256-bit Serpent reduced to rounds $i = 1, \ldots, 8$ (or rounds $i = 9, \ldots, 16$ or rounds $i = 17, \ldots, 24$). The basic idea is that we peel off the last round by guessing the last round subkey and look for our property in the preceding seven rounds.

A difficulty arises because the boomerang attack makes adaptive chosen plaintext *and* ciphertext queries. Suppose we encrypt $P$ and $P'$ to get $C$ and

$C'$. To get $D$ and $D'$, we must peel off one round from each ciphertext $C$ and $C'$, XOR the result with $\nabla$, and then re-encrypt the last round with the guessed subkey. To do this, we will have to guess the 68 bits of the last round subkey corresponding to the 17 active S-boxes of $B'_8$. Assume we consider $2^{94}$ plaintext pairs $P$ and $P'$. For each of these pairs, we will have to compute $2^{68}$ different pairs $Q$ and $Q'$ (for each of the $2^{68}$ possible last round subkeys). Unfortunately, this means we will likely end up working with the entire codebook of all $2^{128}$ possible plaintext/ciphertext pairs.

If we are willing to work with the entire codebook of $2^{128}$ plaintexts and ciphertexts, then we can extract the last round subkey in the following manner. We request the ciphertexts $C$ and $C'$ of $2^{96}$ plaintext pairs with an input difference $\Delta$. Then for each of our $2^{68}$ possible last round subkeys and for each of our $2^{96}$ ciphertext pairs, we compute the boomerang ciphertexts $D$ and $D'$. We then request the plaintexts $Q$ and $Q'$ corresponding to these ciphertexts. If we correctly guess the last round subkey, we should expect to see the plaintext difference $Q \oplus Q' = \Delta$ with probability $2^{-94}$. That is, for the correct subkey we should expect to see the difference $Q \oplus Q' = \Delta$ approximately four times. (Or, put yet another way, if we guess the correct subkey, we should generate about four right quartets.)

This attack requires $2^{68} \times 2^{97}$ partial decryptions and encryptions, or approximately $2^{163}$ eight-round Serpent encryptions. This attack also requires access to the entire codebook, and thus $2^{128}$ plaintexts and $2^{133}$ bytes of random-access memory.

## 5 Amplified Boomerang Attacks

In [KKS00] we introduced a new class of cryptanalytic attacks which we call "amplified boomerangs." Amplified boomerang attacks are similar to traditional boomerang attacks but require only chosen plaintexts. The chosen-plaintext–only requirement makes the amplified boomerang attacks more practical than the traditional boomerang attacks in many situations. In [KKS00] we describe a seven-round boomerang amplifier distinguishing attack and an eight-round boomerang amplifier key recovery attack requiring $2^{113}$ chosen plaintext pairs, $2^{119}$ bytes of random-access memory, and roughly $2^{179}$ Serpent eight-round encryptions.

### 5.1 Amplified Seven-Round Distinguisher

In this section we review the seven-round amplified boomerang distinguishing attack presented in [KKS00]. We request $2^{112}$ plaintext pairs with our input difference $\Delta$. After encrypting with the first half of the cipher $E_0$, we expect roughly $2^{81}$ pairs to satisfy the first characteristic $\Delta \rightarrow \Delta^*$. There are approximately $2^{161}$ ways to form quartets using these $2^{81}$ pairs. We expect there to be approximately $2^{33}$ quartets $(Y_4^0, Y_4^1)$ and $(Y_4^2, Y_4^3)$ such that $Y_4^0 \oplus Y_4^2 = \nabla^*$. However, because $(Y_4^0, Y_4^1)$ and $(Y_4^2, Y_4^3)$ are right pairs for the first half of the

cipher, and $Y_4^0 \oplus Y_4^1 = Y_4^2 \oplus Y_4^3 = \Delta^*$, we have that $Y_4^1 \oplus Y_4^3$ must also equal $\nabla^*$. In effect, the randomly occurring difference between $Y_4^0$ and $Y_4^2$ has been "amplified" to include $Y_4^1$ and $Y_4^3$.

At the input to $E_1$ we expect approximately $2^{33}$ quartets with a difference of $(\nabla^*, \nabla^*)$ between the pairs. This gives us approximately two quartets after the seventh round with an output difference of $(\nabla, \nabla)$ across the pairs. We can identify these quartets by intelligently hashing our original ciphertext pairs with our ciphertext pairs XORed with $(\nabla, \nabla)$ and noting those pairs that collide. For a random distribution, the probability of observing a single instance of our cross-pair difference $(\nabla, \nabla)$ is approximately $2^{-33}$.

### 5.2 Amplified Eight-Round Key Recovery Attack

In [KKS00] we extended the previous distinguishing attack to an eight-round key-recovery attack on rounds one through eight of Serpent requiring $2^{113}$ chosen-plaintext pairs, $2^{119}$ bytes of random-access memory, and work equivalent to approximately $2^{179}$ eight round Serpent encryptions. In this attack we guess 68 bits of Serpent's last round key $K_9$. For each key guess, we peel off the last round and perform the previous distinguishing attack.

### 5.3 Experimental Improvements to the Eight-Round Attack

We can improve our eight-round boomerang amplifier attack by observing that we do not need to restrict ourselves to using only one specific cross-pair difference $(\nabla^*, \nabla^*)$ after $E_0$. That is, rather than considering only pairs of pairs with a cross-pair difference of $(\nabla^*, \nabla^*)$ after $E_0$, we can use pairs of pairs with a cross-pair difference of $(x, x)$ after $E_0$, for *any* $x$, provided that both pairs follow the characteristic $x \rightarrow \nabla$ through $E_1$ with sufficiently high probability.

Experimentally, we find that $\sum_x Pr[x \rightarrow \nabla \text{ through } E_1]^2$ is approximately $2^{-23}$.[3] Consequently, if we request $2^{109}$ chosen-plaintext pairs with our input difference $\Delta$ to $E_0$, we should expect approximately 16 pairs of pairs with a cross-pair difference of $(\nabla, \nabla)$ after $E_1$. This reduces the work of our attack in Section 5.2 to approximately $2^{175}$ eight-round Serpent encryptions. As noted in [Wag99], this observation can also be used to improve the standard boomerang attack.

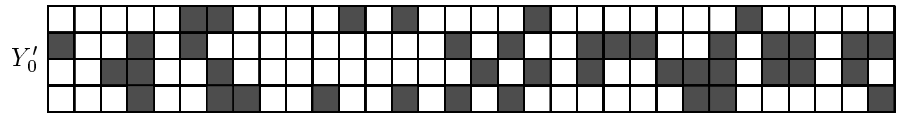
### 5.4 Amplified Nine-Round Key Recovery Attack

We can further extend the above eight-round attack to break nine rounds of 256-bit Serpent using less work than exhaustive search. To do this, let us consider a nine-round Serpent variant corresponding to rounds zero through eight of

[3] We generated $2^{28}$ pairs of ciphertext pairs with a cross-pair difference $(\nabla, \nabla)$. We decrypted each pair through $E_1^{-1}$ and counted the number of pairs with a cross pair difference $(x, x)$ for any $x$. We observed 35 such pairs of pairs.

Serpent. Let us still refer to rounds one through four as $E_0$ and rounds five through seven as $E_1$.

If we apply the inverse linear transformation to $\Delta$ we get




where $Y_0'$ has 24 active S-boxes. We request $2^{14}$ blocks of $2^{96}$ chosen plaintexts such that each block varies over all the possible inputs to the active S-boxes in $Y_0'$. This gives us $2^{109}$ pairs with our desired difference $\Delta$ into $E_0$ and 16 pairs of pairs with a cross-pair difference $(\nabla, \nabla)$ after $E_1$. In order to identify our amplified boomerang, we must guess 96 bits of the first round subkey $K_0$ and 68 bits of the last round subkey $K_9$.

We first guess the 96 bits of $K_0$ corresponding to the 24 active S-boxes in $Y_0'$. For each 96 bit key guess and for each plaintext $P$, we encrypt $P$ one round to $Y_0$. We store $P$ with satellite data $Y_0$ in HASH0$[K_0]$ and we store $Y_0$ with satellite data $P$ in HASH1$[K_0]$. This step takes approximately $2^{212}$ bytes of random-access memory and work equivalent to $2^{203}$ Serpent eight-round encryptions.

Next, for each 68-bit key guess of $K_9$, we want to establish a list of all pairs $(P^0, P^2)$ that have difference $\nabla$ as the output of the eighth round. To do this, for each ciphertext $C^0$, we decrypt up one round to $X_8^0$, compute $X_8^2 = X_8^0 \oplus B_8'$, and store $(X_8^0, X_8^2)$ or $(X_8^2, X_8^0)$ in a hash table (where the order of $X_8^0$ and $X_8^2$ depends on whether $X_8^0$ is less than $X_8^2$). The satellite data in our hash table entry includes the plaintext $P^0$ corresponding to $C^0$. If a collision occurs in our hash table, we have found two plaintexts $P^0$ and $P^2$ that have our desired difference $\nabla$ after the eighth round. We store these pairs $(P^0, P^2)$ in LIST2$[K_9]$ and HASH2$[K_9]$. This step takes approximately $2^{184}$ bytes of random-access memory and work equivalent to $2^{175}$ Serpent eight-round encryptions.

The following algorithm counts the number of occurrences of our boomerang amplifier through $E_1 \circ E_0$. This algorithm can be thought of as sending a boomerang from the ciphertext to the plaintext and back again:

**for** each 96-bit subkey guess of $K_0$ **do**
  **for** each 68-bit subkey guess of $K_9$ **do**
    $count \leftarrow 0$
    **for** each pair $(P^0, P^2)$ in LIST2$[K_9]$ **do**
      lookup $Y_0^0$, $Y_0^2$ corresponding to $P^0$, $P^2$ in HASH0$[K_0]$
      $Y_0^1 \leftarrow Y_0^0 \oplus Y_0'$, $Y_0^3 \leftarrow Y_0^2 \oplus Y_0'$
      lookup $P^1$, $P^3$ corresponding to $Y_0^1$, $Y_0^3$ in HASH1$[K_0]$
      **if** $(P^1, P^3)$ in HASH2$[K_9]$ **then**
        $count \leftarrow count + 1$
    **if** $count \geq 15$ **then**
      save key guess for $K_0$, $K_9$

For each subkey guess guess of $K_9$, we expect LIST2$[K_9]$ will contain approximately $2^{219} \times 2^{-128} = 2^{91}$ pairs. Consequently, we expect the inner loop of the

above algorithm to execute $2^{255}$ times. This attack requires work equivalent to approximately $2^{252}$ Serpent nine-round encryptions.

## 6 Meet-in-the-Middle Attacks

Although not as powerful as our previous attacks, we can use the meet-in-the-middle technique to attack six-round Serpent. In the meet-in-the-middle attack, we try to determine the value of a set of intermediate bits in a cipher by guessing key bits from both the plaintext and ciphertext sides. The attack looks for key guesses that match on the predicted values of the intermediate bits.

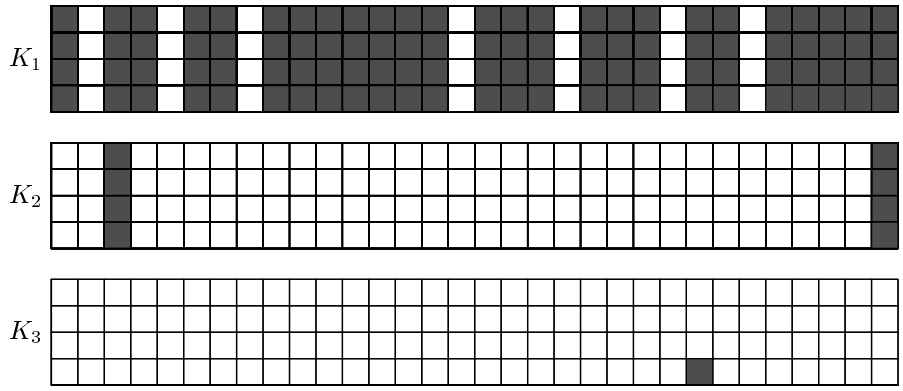
We did a computer search for the best meet-in-the-middle attacks that isolate a set of bits in one column of an intermediate state of Serpent. Table 2 summarizes our results. Although we can also use the meet-in-the-middle technique to predict bits in more than one column of an intermediate state of Serpent, doing so requires additional key guesses and is thus undesirable.

| Rounds | $b$ | $s$ | Key guess from top | Key guess from bottom |
|---|---|---|---|---|
| 6 | 1 | $B_3$ | 236 | 239 |
| 5 | 2 | $B_2$ | 152 | 223 |
| 5 | 3 | $B_2$ | 176 | 224 |
| 5 | 4 | $B_2$ | 204 | 225 |
| 6 | 1 | $X_3$ | 237 | 238 |
| 5 | 2 | $X_2$ | 154 | 221 |
| 5 | 3 | $X_2$ | 179 | 221 |
| 5 | 4 | $X_2$ | 208 | 221 |
| 5 | 1 | $Y_2$ | 200 | 104 |
| 5 | 2 | $Y_2$ | 200 | 178 |
| 5 | 3 | $Y_2$ | 208 | 198 |
| 5 | 4 | $Y_2$ | 208 | 221 |

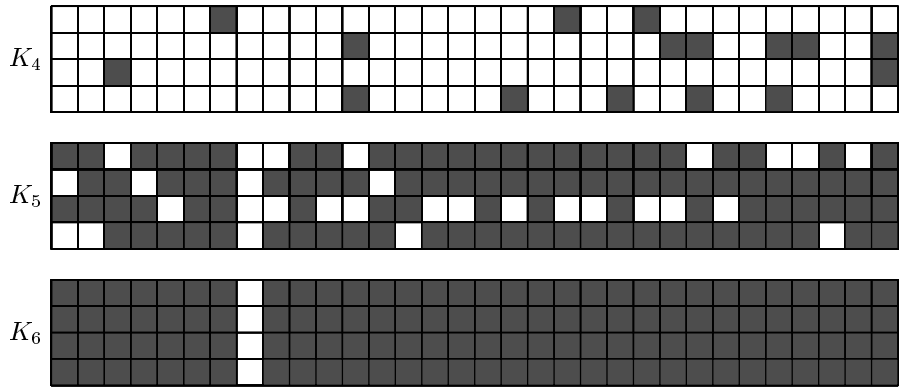
**Table 2.** Meet-in-the-middle requirements to determine $b$ intermediate bits of internal state $s$ in a given number round Serpent variant.

The clearest way to illustrate the meet-in-the-middle attack on Serpent is through diagrams similar to those used in Section 3 and Appendix A. The plaintext in this attack on six-round Serpent is $B_0$ and the ciphertext is $B_6$. The bit we are trying to predict is the eighth most significant bits of $x_3$ where $x_3$ is the fourth word of $X_3$, $X_3 = (x_0, x_1, x_2, x_3)$.

The 237 key bits guessed from the plaintext side are




and the 238 key bits guessed from the ciphertext side are




where the shaded cells denote the bits we guess.

The attack proceeds as follows. We obtain 512 known plaintexts and their corresponding ciphertexts. For each plaintext key guess, we compute the target bit of $X_3$ for each of our 512 plaintexts. We concatenate these bits for each plaintext into a 512-bit value. We then store this 512-bit value, along with the associated key guess, in a hash table.

For each ciphertext key guess, we proceed along the same lines and compute the target bit of $X_3$ for each of our 512 ciphertexts. We concatenate these bits for each ciphertext into a 512-bit value and look for this value in our hash table. If we find such a value, then the plaintext and ciphertext keys suggested by the match will likely be correct. This attack requires approximately $2^{246}$ bytes of random-access memory and work equivalent to $2^{247}$ six-round encryptions.

## 7 Key Schedule Observations

This section addresses some observations we have about the Serpent key schedule. We currently do not know of any cryptanalytic attacks that use these observations.

As described in Section 2, the prekeys $w_0, w_1, \ldots, w_{131}$ are computed using the recurrence

$$w_i \leftarrow (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) \lll 11 \tag{1}$$

where $w_{-8}, \ldots, w_{-1}$ is the initial 256 bit master key. If we ignore the rotation and the internal XOR with $\phi$ and $i$, we get the linear feedback construction

$$w_i \leftarrow w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \tag{2}$$

Let us now consider two keys $K$ and $K^*$ that have a difference $K' = K \oplus K^*$. The prekeys for $K$ and $K^*$ expand to $w_0, \ldots, w_{131}$ and $w_0^*, \ldots, w_{131}^*$, respectively. By virtue of Equation 2, the prekey differences for $K'$ can be computed using the recurrence

$$w_i' = w_i \oplus w_i^* = w_{i-8}' \oplus w_{i-5}' \oplus w_{i-3}' \oplus w_{i-1}' \tag{3}$$

for $i = 0, \ldots, 131$. If we use the original recurrence (Equation 1) to compute the prekeys rather than Equation 2, the recurrence for $w_i'$ becomes

$$w_i' = (w_{i-8}' \oplus w_{i-5}' \oplus w_{i-3}' \oplus w_{i-1}') \lll 11 \tag{4}$$

for $i = 0, \ldots, 131$.

For any key $K$, the $i$th round subkey $K_i$ is computed from the four prekeys $w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}$. The same can be said for the key $K^*$. If for any given round $i$ the four prekeys for $K$ are equivalent to the corresponding four prekeys for $K^*$, then the subkeys $K_i$ and $K_i^*$ will be equivalent; this occurs when the prekey differences $w_{4i}', w_{4i+1}', w_{4i+2}', w_{4i+3}'$ are zero.

Let us now observe some situations where the prekey differences for the $i$th round subkey are zero. As a simple example, let us consider Figure 1. The shaded cells in Figure 1 depict prekeys that are different for $K$ and $K^*$. The unshaded areas are equivalent between the keys. Notice that six out of the 33 128-bit subkeys are equivalent.

There is a heavy restriction on Figure 1: all the differences must be the same. That is, when Equation 2 is used for the prekey computation, it must be that $w_{-5}' = w_{-3}' = w_{-1}' = \cdots = w_{127}' = k$ for some constant $k$. If we consider the original prekey recursion (Equation 4), this example works only when $k =$ `0xFFFFFFFF`. Furthermore, when the non-zero prekey differences are `0xFFFFFFFF`, six out of 33 subkeys are equivalent and five out of 33 subkeys have complementary prekeys.

## 8 Conclusions

In this paper we consider several attacks on Serpent. We show how to use differential, boomerang, and amplified boomerang techniques to recover the key for Serpent up to nine rounds. We also show how to break six rounds of Serpent

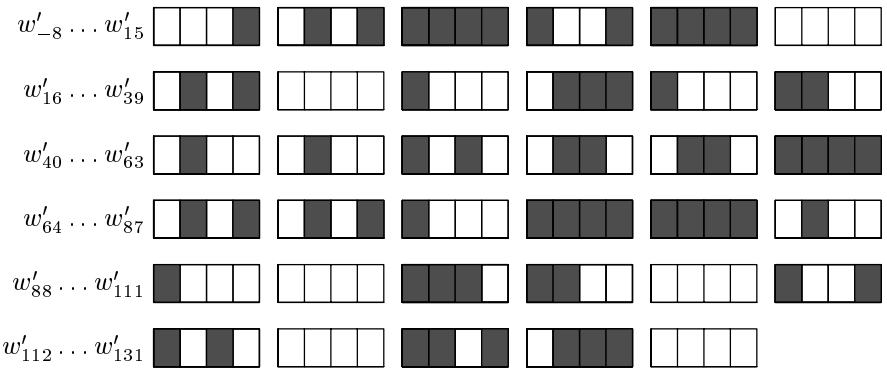



**Fig. 1.** Difference propagation in the key schedule when $w'_{-5} = w'_{-3} = w'_{-1} =$ `0xFFFFFFFF`.

using a meet-in-the-middle attack. We then provide key schedule observations that may someday be used as the foundation for additional attacks.

Although these attacks do not come close to breaking the full 32-round cipher, we feel that these results are worth reporting for several reasons. Specifically, the results and observations in this paper provide a starting point for additional research on Serpent. These results also provide a security reference point for discussions about modifying the number of rounds in Serpent.

In conjunction with the previous observation, we would like to point out that there are several avenues for further research. Although our current paper addresses differential attacks against Serpent, we have not yet tried linear and differential-linear attacks. We are also attempting to mount additional boomerang variants against Serpent. We expect that all these attacks, while quite capable of breaking reduced-round versions of Serpent, will fail to break the entire 32-round Serpent. In order to break a substantial portion of Serpent's 32 rounds, we suspect that entirely new attacks may need to be invented.

## 9 Acknowledgements

The "extended Twofish team" met for two week-long cryptanalysis retreats during Fall 1999, once in San Jose and later in San Diego. This paper is a result of those collaborations. Our analysis of Serpent has very much been a team effort, with everybody commenting on all aspects. The authors would like to thank Niels Ferguson, Chris Hall, Mike Stay, David Wagner, and Doug Whiting for useful conversations and comments on these attacks, and for the great time we had together.

## References


[ABK98] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, 1998.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[Dun99] O. Dunkelman, "An Analysis of Serpent-p and Serpent-p-ns," rump session, *Second AES Candidate Conference*, 1999.

[KKS00] J. Kelsey, T. Kohno, and B. Schneier, "Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent," *Fast Software Encryption, 7th International Workshop*, to appear.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions," *Second AES Candidate Conference*, 1999.

[Wag99] D. Wagner, "The Boomerang Attack," *Fast Software Encryption, 6th International Workshop*, Springer-Verlag, 1999.

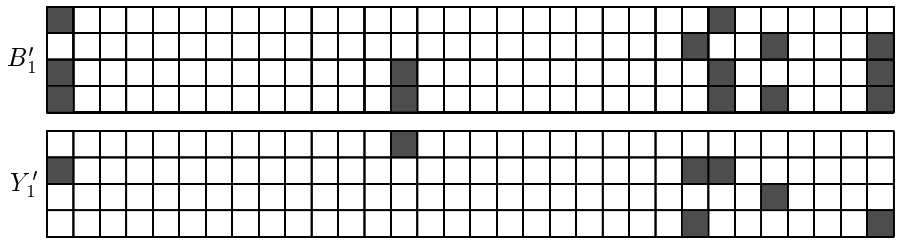

## A Differential Characteristics

### A.1 Five-Round Characteristic

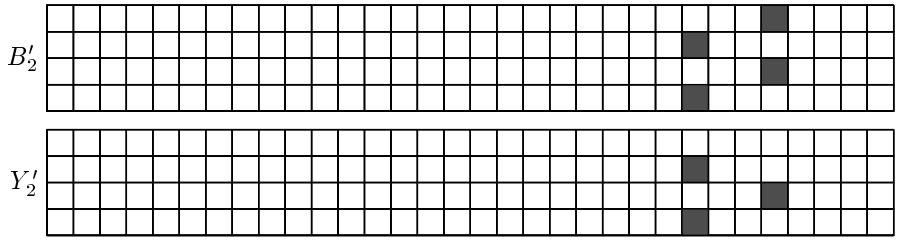
The following is an example of a five-round differential characteristic with probability $p = 2^{-80}$. We used this characteristic in Section 3. This characteristic passes between rounds $i = 1 \bmod 8$ and $i = 5 \bmod 8$. We used only the first four rounds of this five-round characteristic for our boomerang attack in Section 4.

We illustrate this characteristic by showing five one-round characteristics that can be connected with the Serpent linear transformation $L$. The shaded bits in the figures denote differences in the pairs. We feel that these figures provide an intuitive way to express Serpent's internal states.

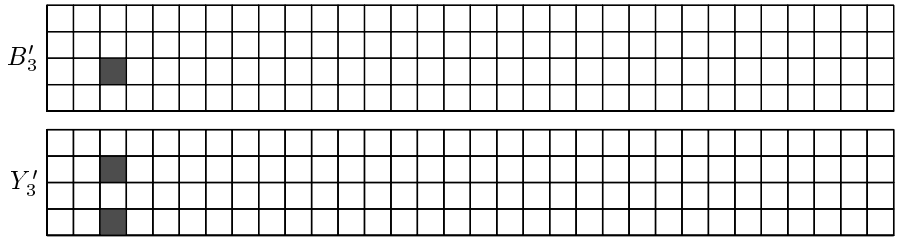
The first-round characteristic, $B'_1 \rightarrow Y'_1$, has probability $2^{-13}$:



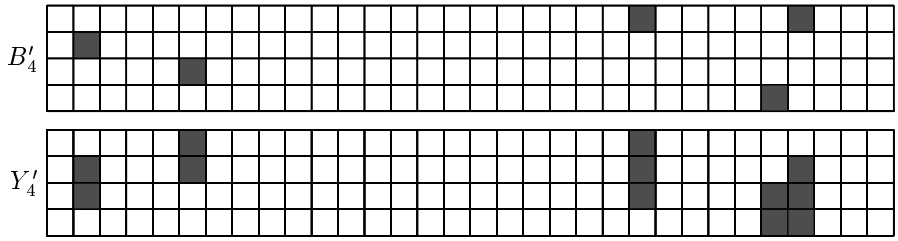

The second-round characteristic has probability $2^{-5}$:



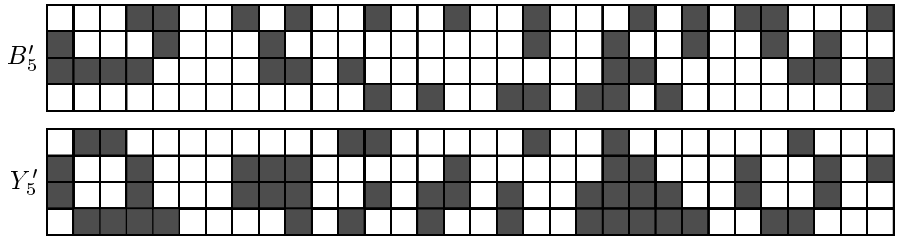
The third-round characteristic has probability $2^{-3}$:




The fourth-round characteristic has probability $2^{-10}$.



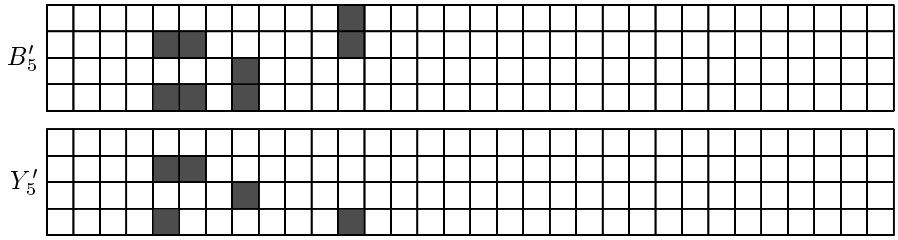

The fifth-round characteristic has probability $2^{-49}$.



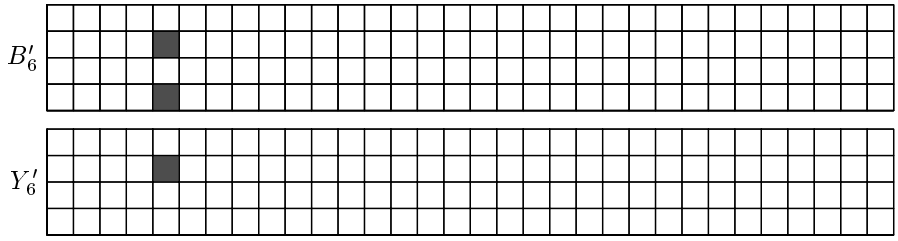

### A.2 Boomerang Characteristic

The following is an example of a three-round characteristic with probability $p = 2^{-16}$. We used this characteristic in Section 4. This characteristic passes between rounds $i = 5 \bmod 8$ and $i = 7 \bmod 8$.

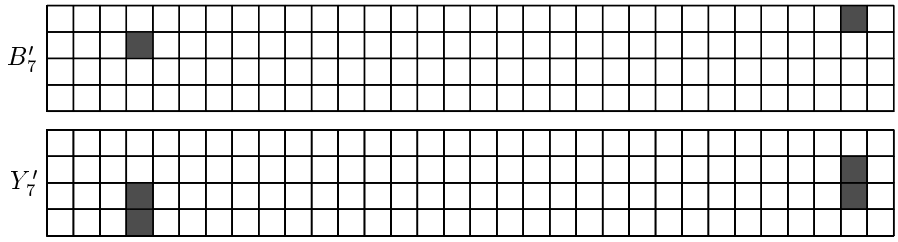
The fifth-round characteristic has probability $2^{-10}$:



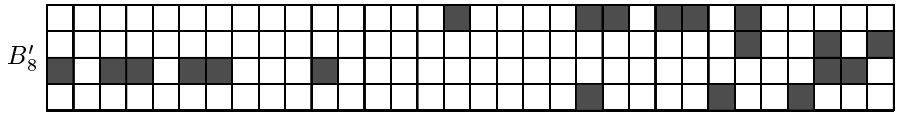

The sixth-round characteristic has probability $2^{-2}$:




The seventh-round characteristic has probability $2^{-4}$:




If we apply the linear transformation $L$ to $Y'_7$, we get:



# *Session 5:*

# *"Cryptographic Analysis and Properties"*

# *(II)*

# Attacking Seven Rounds of Rijndael under 192-bit and 256-bit Keys

Stefan Lucks*

Theoretische Informatik
University of Mannheim, 68131 Mannheim, Germany
lucks@th.informatik.uni-mannheim.de

**Abstract.** The authors of Rijndael [3] describe the "Square attack" as the best known attack against the block cipher Rijndael. If the key size is 128 bit, the attack is faster than exhaustive search for up to six rounds. We extend the Square attack on Rijndael variants with larger keys of 192 bit and 256 bit. Our attacks exploit minor weaknesses of the Rijndael key schedule and are faster than exhaustive search for up to seven rounds of Rijndael.

## 1 Introduction

The block cipher Rijndael [3] has been proposed as an AES candidate and was selected for the secound round. It is a member of a fast-growing family of Square-like ciphers [2–4, 6, 7].

Rijndael allows both a variable block length of $M * 32$ bit with $M \in \{4, 6, 8\}$ and a variable key length of $N * 32$ bit, $N$ an integer. In the context of this paper we concentrate on $M = 4$, i.e., on a block length of 128 bit, and on $N \in \{4, 6, 8\}$, i.e., on key sizes of 128, 192, and 256 bit. We abridge these variants by RD-128, RD-192 and RD-256. The number $R$ of rounds is specified to be $R = 10$ for RD-128, $R = 12$ for RD-192, and $R = 14$ for RD-256. In the context of this paper, we consider reduced-round versions with $R \leq 7$.

The authors of Square [2] described the "Square attack", a dedicated attack exploiting the byte-oriented structure of Square. The attack works for Square reduced to six rounds and is applicable to Rijndael and other Square-like ciphers as well [3, 4, 1]. This paper deals with extensions of the Square-attack for RD-192 and RD-256.

In Section 2, we shortly describe Rijndael, leaving out many details and pointing out some properties relevant for our analysis. Section 3 deals with the Square attack for up to six rounds of Rijndael, originating from

* Supported by DFG grant Kr 1521/3-1.

[2, 3]. In Sections 4–6 we describe attacks for seven rounds of Rijndael. The attack in Section 4 and its analysis is valid for all versions of Rijndael, while the attacks in Section 5 and Section 6 are dedicatedly for Rijndael-256 and Rijndael-192, exploiting minor weaknesses of the Rijndael key schedule. We give final comments and conclude in Section 7.

## 2 A Description of Rijndael

Rijndael is a byte-oriented iterated block cipher. The plaintext (a 128-bit value) is used as initial state, the state undergoes a couple of key-dependent transformations, and the final state is taken as the ciphertext. A state $A \in \{0,1\}^{128}$ is regarded as a $4 \times 4$ matrix $(A_{i,j})$, $i, j \in \{0,1,2,3\}$ of bytes (see Figure 1). The four columns of $A$ are $A_i = (A_{0,j}, A_{1,j}, A_{2,j}, A_{3,j})$.

| | | | |
|---|---|---|---|
| (0,0) | (0,1) | (0,2) | (0,3) |
| (1,0) | (1,1) | (1,2) | (1,3) |
| (2,0) | (2,1) | (2,2) | (2,3) |
| (3,0) | (3,1) | (3,2) | (3,3) |

**Fig. 1.** The index positions $(i, j)$ for a 4*4 matrix of bytes.

Given the initial state, $R$ rounds of transformations are applied. Each round can be divided into several elementary transformations.

By the *key schedule*, the key, a $(N*32)$-bit value with $N \in \{4, 6, 8\}$, is expanded into an array $W[\cdot]$ of $4(R+1)$ 32-bit words $W[0], \ldots, W[4(R+1)-1]$. Four such words $W[4r+j]$ with $j \in \{0,1,2,3\}$ together are used as $r$-th "round key" $K^r$, with $r \in \{0, \ldots, R\}$. Like the state, we regard a round key $K^r$ as a $4*4$ matrix of bytes $K^r_{i,j}$ with four columns $K^r_j = W[4r+j]$ for $j \in \{0,1,2,3\}$.

### 2.1 The Elementary Transformations of Rijndael

Rijndael uses four elementary operations to transform a state $A = (A_{i,j})$ into a new state $B = (B_{i,j})$, see also Figure 2:

1. The *byte substitution* (BS): $B_{i,j} := S(A_{i,j})$ for $i, j \in \{0, 1, 2, 3\}$. Here, $S$ denotes a permutation over $\{0, 1\}^8$, i.e., $S^{-1}$ is defined with $A_{i,j} = S^{-1}(B_{i,j})$.
2. The *shift row* operation (SR), a cyclic shift of bytes: $B_{i,j} := A_{i,(j+i) \bmod 4}$.
3. The *mix column* transformation (MC). Each column $A_i$ of state $A$ is transformed via a linear transformation $\mu$ over $\{0, 1\}^{32}$, i.e. $B_i := \mu(A_i)$ for $i \in \{0, 1, 2, 3\}$. Also, $\mu$ is invertible.
   An input $X \in \{0, 1\}^{32}$ for $\mu$ can be seen as a vector $X = (X_0, X_1, X_2, X_3)$ of four bytes. Consider $X' = (X'_0, X'_1, X'_2, X'_3)$ to be different from $X$ in exactly $k$ bytes ($1 \leq k \leq 4$), i.e.
   $$k \;=\; \left|\{\; i \in \{0, 1, 2, 3\} \mid X_i \neq X'_i \;\}\right|.$$
   Then $Y = \mu(X)$ and $Y' = \mu(X')$ are different in at least $5 - k$ of their four bytes. The same property holds for the inverse $\mu^{-1}$ of $\mu$.
4. The *key addition* (KA). The $r$-th round key $K^r = (K^r_{i,j})$ is added to the state $A$ by bit-wise XOR: $B_{i,j} := A_{i,j} \oplus K^r_{i,j}$.

Note that all elementary transformations of Rijndael are invertible.

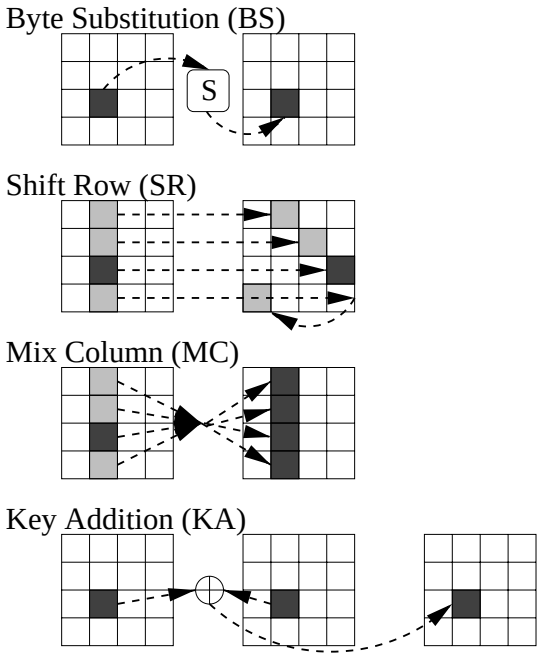



**Fig. 2.** The four elementary transformations of Rijndael.

### 2.2 The Rijndael Round Transformation

For $r \in \{0, \ldots, R\}$, the round key $K^r$ consists of the expanded key words $W[4r], \ldots, W[4r+3]$. The structure of Rijndael is defined as follows[1]:

**1.** $S :=$ plaintext;
**2.** KA ($S$, $K^0$); ($*$ add round key 0 before the first round $*$)
**3.** for $r := 1$ to $R$ do: ($*$ run through round 1, 2, ..., $R$ $*$)
  **4.** $S :=$ BS($S$); ($*$ byte substitution $*$)
  **5.** $S :=$ SR($S$); ($*$ shift row $*$)
  **6.** $S :=$ MC($S$); ($*$ mix column $*$)
  **7.** $S :=$ KA($S$, $K^r$); ($*$ add round key $r$ $*$)
**8.** ciphertext $:= S$.

Steps 4–7 are the "standard representation" of the Rijndael round structure. The implementor of Rijndael has a great degree of freedom to change the order the elementary operations are done – without changing the behavior of the cipher. (We refer the reader to the description of the "algebraic properties" and the "equivalent inverse cipher structure" for details [3, Section 5.3].) We describe one alternative representation of the round structure. As an "alias" for the $r$-th round key $K^r$ we use the value

$$L^r = \mathrm{SR}^{-1}(\mathrm{MC}^{-1}(K^r)). \tag{1}$$

Accordingly, we distinguish between the "$L$-representation" $L^r$ of a round key and its "$K$-representation". Knowing $L^r$ is equivalent to knowing $K^r$, and knowing a column $K^r_j$ of $K^r$ is equivalent to knowing four bytes of $L^r$, see Table 1.

| known column of $K^r$ | known bytes $L^r_{i,j}$ of $L^r$ |
|---|---|
| $K^r_0$ | $(i,j) \in \{(0,0),(1,3),(2,2),(3,1)\}$ |
| $K^r_1$ | $(i,j) \in \{(0,1),(1,0),(2,3),(3,2)\}$ |
| $K^r_2$ | $(i,j) \in \{(0,2),(1,1),(2,0),(3,3)\}$ |
| $K^r_3$ | $(i,j) \in \{(0,3),(1,2),(2,1),(3,0)\}$ |

**Table 1.** Known columns of a key in $K$-representation and the corresponding known key bytes in $L$-representation.

[1] Actually, the authors of Rijndael [3] specify an exception: in the last round, the MC-operation is left out. As was stressed in [3], this modification does not strengthen or weaken the cipher. In the current paper, we assume for simplicity that the last round behaves exactly like the other rounds.

The following describes a functionally equivalent round structure for Rijndael, see also Figure 3.

4. $S := \mathrm{BS}(S)$; ($*$ byte substitution $*$)
5. $S := \mathrm{KA}(S, L^r)$; ($*$ add round key, given in $L$- representation $*$)
6. $S := \mathrm{SR}(S)$; ($*$ shift row $*$)
7. $S := \mathrm{MC}(S)$; ($*$ mix column $*$)

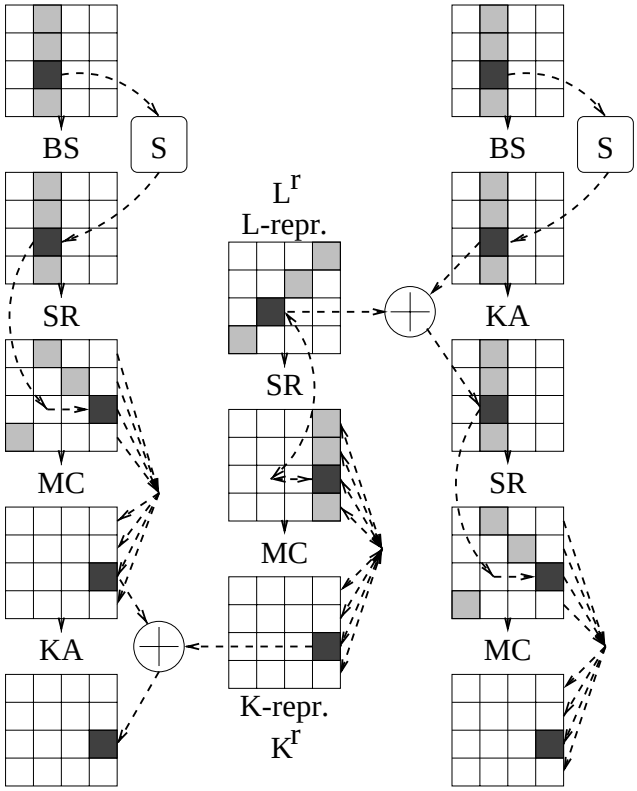



**Fig. 3.** The Structure of a Rijndael round.
Left: The standard representation of the Rijndael round transformation
Middle: The round key – changing between $K$-representation and $L$-representation
Right: The alternative representation of the Rijndael round transformation

### 2.3 The Rijndael Key Schedule

The key schedule is used to generate an expanded key from a short (128–256 bit) "cipher key". We describe the key-schedule using word-wise oper-

ations (where a word is a 32-bit quantity), instead of byte-wise ones. The cipher key consists of $N$ 32-bit words, the expanded key of $4*(R+1)$ such words $W[\cdot]$. The first $N$ words $W[0], \ldots, W[N-1]$ are directly initialised by the $N$ words of the cipher key.

For $k \in \{1, 2, \ldots\}$, const$(k)$ denotes fixed constants, and $f, g : \{0,1\}^{32} \to \{0,1\}^{32}$ are nonlinear permutations.[2] For $i \in \{N, \ldots, 4*(R+1)-1\}$ the words $W[i]$ are defined recursively:

$$\begin{array}{l} \text{If } (i \bmod N) = 0 \\ \quad \text{then } W[i] := W[i-N] \oplus f(W[i-1]) \oplus \text{const}(i \operatorname{div} N) \\ \quad \text{else if } ((N > 6) \text{ and } (i \bmod N) = 4) \\ \qquad \text{then } W[i] := W[i-N] \oplus g(W[i-1]) \\ \qquad \text{else } W[i] := W[i-N] \oplus W[i-1]. \end{array} \tag{2}$$

Note that two words $W[i-1]$ and $W[i-N]$ suffice to compute the word $W[i]$. Similarly, we can go backwards: Given two words $W[i]$ and $W[i-1]$, we can compute $W[i-N]$. (This will be useful for our attacks below.) Hence, any $N$ consecutive words $W[k], \ldots, W[k+N-1]$ of the expanded key suffice to efficiently generate the complete expanded key and thus to completely break Rijndael.

## 3 The Square Attack for Rijndael

In this section we describe the dedicated Square-Attack for Rijndael. More details can be found in [2, 3]. We start with a simple attack on four rounds and extend the simple attack by an additional round at the beginning and another one at the end. This leads to the "Square-6" attack for six rounds of Rijndael. Analysing the performance of our attacks with respect to RD-192 and RD-256 is delayed until the end of this section.

### 3.1 Attacking Four Rounds – the Simple Attack

To describe the attack we need the notion of a "$\Lambda$-set", i.e., a set of $2^8$ states that are all different in some of their $4*4$ bytes (the "active" bytes), and all equal in the other ("passive") bytes. In other words, for two distinct states $A$ and $B$ in a $\Lambda$-set we always have

$$\begin{array}{ll} A_{i,j} \neq A_{i,j} & \text{if the byte at position } (i,j) \text{ is active, and} \\ A_{i,j} = B_{i,j} & \text{else, i.e., if the byte at } (i,j) \text{ is passive.} \end{array}$$

[2] We omit the definition of $f$ and $g$, but we point out that the four functions $f$, $g$, $f^{-1}$ and $g^{-1}$ are fixed in the definition of Rijndael and can be computed efficiently.

A $\Lambda$-set with exactly $k$ active bytes is a "$\Lambda^k$-set".

The adversary chooses one $\Lambda^1$-set $P_0$ of states (plaintexts). By $P_i$ we denote the sets of $2^8$ states which are the output of round $i$. $P_1$ is a $\Lambda^4$-set, all four active bytes in the the same column. $P_2$ is a $\Lambda^{16}$-set. $P_3$ is unlikely to be a $\Lambda$-set. But, as explained in [2, 3], all the bytes of $S_3$ are "balanced", i.e., the following property holds:

$$\text{For all } (i,j) \in \{0,1,2,3\}^2 : \bigoplus_{A \in P_3} A_{i,j} = 0. \tag{3}$$

Recall that we consider a four-round attack, i.e., $P_4$ is the set of $2^8$ ciphertexts the adversary learns. It is unlikely that the bytes of $P_4$ are balanced, but the balancedness of the bytes of $P_3$ can be exploited to find the fourth round key $K^4$. Let $L^4$ be the $L$-representation of $K^4$, cf. Equation (1). The attack defines a set $Q_4$ "in between"[3] $P_3$ and $P_4$:

1. For $X \in P_4$:
   $Y := \mathrm{MC}^{-1}(X)$;
   $Z := \mathrm{SR}^{-1}(Y)$.
   Denote the set of $2^8$ states $Z$ by $Q_4$.
2. For all $(i,j) \in \{0,1,2,3\}^2$:
   for $a \in \{0,1\}^8$:
   $b(a) := \bigoplus_{Z \in Q_4} S^{-1}(Z_{i,j} \oplus a)$;
   if $b(a) \neq 0$ then conclude $L^4_{i,j} \neq a$.

In short, we invert round four step by step: invert the mix column operation, invert the shift row operation, add (a possible choice for) the key byte $L^4_{i,j}$ and invert the byte substitution. If the guess $a \in \{0,1\}^8$ for the key $L^4_{i,j}$ is correct, the set of $2^8$ bytes $S^{-1}(Z_{i,j} \oplus a)$ is balanced, i.e., $b(a) = 0$. But if our guess $a'$ for $L^4_{i,j}$ is wrong, we estimate $b(a') = 0$ to hold with only a probability of $2^{-8}$. Thus, on the average two candidates for for each byte $L^4_{i,j}$ are left – the correct byte and a wrong one. We can easily reconstruct an expected number of less than $2^{16}$ candidates for $L^4$.

Each candidate corresponds with a unique choice for the 128-bit cipher key of RD-128. To find the cipher key, we may either choose a second $\Lambda^1$-set of plaintexts, or just use exhaustive search over all key candidates, using the same $2^8$ known pairs of plaintext and ciphertext as before. With overwhelming probability, either approach uniquely determines the

[3] In general, we regard $Q_5$ to be a set of states "in between" $P_{r-1}$ and $P_r$. Note that converting a state in $P_r$ into its counterpart in $P_4$ does not depend on the key and can be just like converting a round key from its $K$-representation into its $L$-representation, similarly to Equation (1).

cipher key, using few memory and an amount of work determined by step 2, i.e., about $2^{20}$ byte-wise XOR-operations. Note that the first approach needs twice as many chosen plaintexts as the second one.

### 3.2 An Extension at the End

As suggested in [2, 3], the above basic attack can be extended by one additional round at the beginning and another round at the end. We start with extending the additional round at the end.

Let $P_0$ be chosen as above. By $P_5$, we denote the set of $2^8$ outputs of round 5. Similar to $Q_4$ above, the adversary can find $Q_5$ by applying $\mathrm{MC}^{-1}$ and applying $\mathrm{SR}^{-1}$. Given the set $Q_5$, we can compute $P_3$ by inverting $1\frac{1}{2}$ rounds of Rijndael.

If the set $P_5$ (or $Q_5$) is fixed, the bytes of $P_3$ at position, say, $(0, 1)$ only depend on $L^5_{0,1}$, $L^5_{1,1}$, $L^5_{2,1}$, $L^5_{3,1}$, and on $L^4_{0,1}$, see Figure 4.

Similar to the four-round attack, we may guess such a five-tuple of key bytes and compute the corresponding bytes of $P_3$. If these aren't balanced, we reject the corresponding key bytes. We expect one out of $2^8$ incorrect five-tuples to be *not* rejected. With five $\Lambda$-sets of plaintexts, i.e., $5 * 2^8$ chosen plaintexts, the the cipher key can easily be found via exhaustive search. (A more diligent treatment would allow us to reduce the number of chosen plaintexts for this attack, but without much effect on the required number of chosen plaintexts for the six-round attack below.)

To measure the running time of our attacks, we use the notion of a "basic operation". Given a column $Y_j$ of bytes of a state $Y$ in $Q_r$, the key column $L^r_j$ and another key byte $L^{r-1}_{k,j}$ with the row index $k$ as the input, we compute the byte $X_{k,j}$ of a state $X$ in $P_{r-2}$, using $V = (V_0, V_1, V_2, V_3)$ and $W = (W_0, W_1, W_2, W_3)$ as intermediate values and define the basic operation $X_{k,j} = \mathrm{BO}(Y_j, k, L^r_j, L^{r-1}_{k,j})$ as follows:

**1.** For $i := 0$ to 3: $V_i := S^{-1}(Y_{i,j} \oplus L^r_{i,j})$.
**2.** $W := \mu^{-1}(V)$.
**3.** $X_{k,j} := S^{-1}(W_{k,j} \oplus L^{r-1}_{k,j})$.

In short: one basic operation requires 5 byte-wise XORs, 5 evaluations of $S^{-1}$, and one evaluation of $\mu^{-1}$.

To check the correctness of a quintuple of bytes, we have to do $2^8$ basic operations and to XOR the results for a balance-check by verifying Equation (3). We do this for every quintuple of bytes. Thus, the five-round attack takes the time of about $2^{48}$ basic operations.

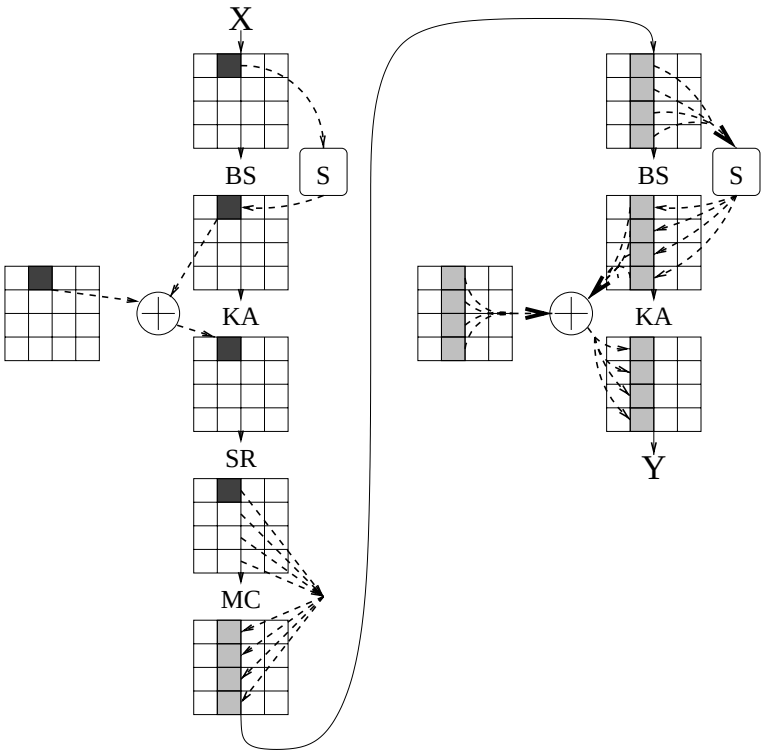



**Fig. 4.** $1\frac{1}{2}$ rounds of Rijndael: Given one column $Y_j$ of the output state $Y$ and five corresponding key bytes, one can find one byte $X_{k,j}$ of the input $X$ by inverting these $1\frac{1}{2}$ rounds of Rijndael; we write $X_{k,j} = \mathrm{BO}(Y_j, k, \textit{key bytes})$ and consider this a "basic operation" for our attacks.

### 3.3 Attacking Six Rounds – the Square-6 attack

Now we extend the above attack by an additional "round 0". We denote this attack the "Square-6 attack".

Let $P_0$ be a $\Lambda^1$-set, as before. By doing one additional round of *de*cryption, we get a $\Lambda^4$-set $P_{-1}$. The active bytes of $P_{-1}$ are at positions determined by the positions of the active $P_0$-byte. E.g., if the $P_0$-byte at position $(0,0)$ is the active one, the $P_{-1}$-bytes at positions $(0,0)$, $(1,3)$, $(2,2)$, and $(3,1)$ are active.

The idea is to arbitrarily fix the plaintext bytes at the passive positions and to choose $2^{32}$ plaintexts varying at the active positions. If the four corresponding bytes of the round key $K^{-1}$ for round 0 are known (e.g. for the active $P_0$-byte at position $(0,0)$: if $K^{-1}_{0,0}$, $K^{-1}_{1,3}$, $K^{-1}_{2,2}$ and

$K_{3,1}^{-1}$ are known), the adversary can easily determine many $2^8$-sets $P_{-1}$ of plaintexts, such that the sets $P_0$ are $\Lambda^1$-sets.

The adversary accordingly chooses $2^{32}$ plaintexts and, for all $2^{32}$ relevant key bytes, runs the five-round attack described above. This is $2^{32}$ times slower than the five-round attack itself, i.e. takes about $2^{80}$ basic operations. The memory requirement for this attack is dominated by the need to store $2^{32}$ ciphertexts.

### 3.4 Considering RD-192 and RD-256

Note that the six-round Square-6 attack and the five-round attack allow us to finds two round keys $K^4$ and $K^5$ at the same time. (The attacker chooses a five-tuple of key bytes, one byte from $K^4$ and four from $K^5$, and probabilistically verify if that choice is correct.) Once the attacker knows two consecutive round keys, i.e. eight consecutive words from the expanded key, the attacker can easily run the key schedule backwards to find the cipher key. In other words, the performance of the Square-6 attack does not depend on which flavor of Rijndael we attack, RD-128, RD-192, or RD-256. We call such an attack a "generic" attack.

The simple four-round attack only provides the attacker with the round key $K^4$. But finding the round key $K^3$ is easy, since the set $P_2$ of states is a $\Lambda^{16}$-set.

## 4 A Generic Attack for Seven Rounds of Rijndael

It is easy to extend the Square-6 attack to seven rounds of Rijndael:

**1.** Choose $2^{32}$ input plaintexts for the Square-6 attack and ask for the corresponding ciphertexts.
**2.** For all $K^7 \in \{0,1\}^{128}$:
  **3.** Last-round-decrypt the $2^{32}$ ciphertexts under $K^7$.
  **4.** Run the Square-6 attack for the results, to get the round keys $K^6$ and $K^5$.
  **5.** Given the round keys $K^5$, $K^6$ and $K^7$, we have more than sufficient key material to recover the complete extended key and to check it for correctness.

The seven-round attack requires the same amount of chosen plaintexts and memory as the Square-6 attack. The running time increases by a factor of $2^{128}$, i.e. to the equivalent of

$$2^{80} * 2^{128} = 2^{208} \text{ basic operations.}$$

Even though the attack is generic, it is pointless for attacking either RD-128 or RD-192 – exhaustive key search is much faster for these variants of Rijndael.

## 5 Attacking Seven Rounds of RD-256

For RD-256, the above generic seven-round attack improves on exhaustive search. But, as shown here, the RD-256 key schedule allows us to accelerate the attack by a factor of $2^8$.

Note that if we know (or have chosen) $K^7$, we know the expanded key words $W[28]$, $W[29]$, $W[30]$, and $W[31]$. By Formula (2), we get

$$\begin{aligned} W[21] &= W[28] \oplus W[29], \\ W[22] &= W[29] \oplus W[30], \text{ and} \\ W[23] &= W[30] \oplus W[31]. \end{aligned}$$

Hence, we know know three columns of $K^5$, including e.g. $K^5_1$. As explained in Section 2.3, this implies knowing 12 bytes of $L^5$, including e.g. $L^5_{0,1}$. To test the bytes of 256-set $P_4$ at position $(0,1)$, we need the bytes in column 1 from $Q_6$, the corresponding key column $L^6_1$ from $L^6$ and the key byte $L^5_{0,1}$ from $L^5$ (cf. Figure 4 at page 9). We attack seven rounds of RD-256 by the following algorithm:

**1.** Choose $2^{32}$ distinct input plaintexts, varying at the byte positions $(0,0)$, $(1,2)$, $(2,2)$, and $(3,1)$ and constant at the other byte positions. Ask for the corresponding ciphertexts.
**2.** For all $2^{32}$ combinations of $K^0_{0,0}$, $K^0_{1,3}$, $K^0_{2,2}$, $K^0_{3,1}$:
  **3.** Fix 32 distinct sets $P_0[i]$ of plaintexts ($i \in \{0,\ldots,31\}$) with $|P_0[i]| = 2^8$, such that the corresponding $P_1[i]$ are $\Lambda^1$-sets.
  **4.** For all $2^{128}$ round keys $K^7$:
    **5.** Decipher the 32 sets of ciphertexts $P_7[i]$ to get $P_6[i]$ and $Q_6[i]$.
    **6.** Compute $L^5_{0,1}$.
    **7.** For all $2^{32}$ combinations of $L^6_1 = (L^6_{0,1}, L^6_{1,1}, L^6_{2,1}, L^6_{3,1})$:
      **8.** $i := 0$; reject := false;
      **9.** while $i \leq 31$ and reject=false:
        begin
          **10.** Compute

$$b[i] := \bigoplus_{A \in Q_6[i]} \mathrm{BO}(A_1, 1, L^6_1, L^5_{0,1}).$$

**11.** If $b[i] = 0$ then i := i+1
else reject := true.
end (∗ while ∗).
**12.** If reject=false then stop (∗ and accept key bytes ∗).

The above algorithm exhaustively searches a subspace of size $2^{192}$ of the full key space. When all 24 key bytes are correct, step 11 always executes then-clause and increments the counter $i$. After 32 such iterations, the algorithm stops in step 12.

If any of the 24 byte key bytes is wrong, we execute the then-clause only with a probability of $2^{-8}$. Since the counter $i$ runs from 0 to 31, the probability for a wrong 24-tuple of key bytes to be accepted is below $2^{-8*32} = 2^{-256}$. There are only $2^{192}$ such tuples of key bytes, thus the probability to accept any wrong 24-tuple is less than $2^{32}*2^{128}*2^{32}*2^{-256} \leq 2^{-64}$, i.e. negligible.

When stopping, the algorithm accepts $K^7$ and four bytes of $K^6$ (or $L^6$). By exhaustive search, it is easy to find the other 12 bytes of $K^6$. Having done that, the key schedule allows to find the full expanded key.

For the attack, $2^{32}$ chosen plaintexts suffice, and the required storage space is dominated by the need to store the corresponding $2^{32}$ ciphertexts.

What about the running time? The loop in step 2 is iterated $2^{32}$ times, step 4 takes $2^{128}$ iterations, and the loop in step 7 is iterated $2^{32}$ times. On the average, the while-loop is iterated $1 + 2^{-8} + 2^{-16} + \ldots$ times, i.e., about once. Step 12 needs $2^8$ basic operations. This makes about

$$2^{32} * 2^{128} * 2^{32} * 1 * 2^8 = 2^{200} \text{ basic operations.}$$

## 6 Attacking Seven Rounds of RD-192

In the case of RD-192, accelerating the generic attack by a factor of $2^8$, as in the case of RD-256, would still not suffice to outperform exhaustive search. Fortunately (for the cryptanalyst), the RD-192 key schedule allows an acceleration by a factor of $2^{24}$, compared to the generic attack,

The columns of $K^7$ are the words $W[28]$, $W[29]$, $W[30]$, and $W[31]$ of the expanded key. These four words allow us to compute three more words – in the case of RD-192, these are $W[23]$, $W[24]$, and $W[25]$, cf. Section 2.3. Two of these words are columns of the round key $K^6$, while the third word is a column of $K^5$: $W[24] = K^6_0$, $W[25] = K^6_1$, and $W[23] = K^5_3$.

From $W[23]$, $W[24]$, and $W[25]$, we can compute three useful key bytes for the attack, for example $L^5_{0,3}$, $L^6_{1,3}$, and $L^6_{2,3}$, cf. Table 1 on page 4. The two remaining key bytes (in our example $L^6_{0,3}$ and $L^6_{3,3}$) still have to be found:

**1.** Choose $2^{32}$ distinct input plaintexts, varying at the byte positions $(0,0)$, $(1,2)$, $(2,2)$, and $(3,1)$ and constant at the other byte positions. Ask for the corresponding ciphertexts.
**2.** For all $2^{32}$ combinations of $K^0_{0,0}$, $K^0_{1,3}$, $K^0_{2,2}$, $K^0_{3,1}$:
  **3.** Fix 32 distinct sets $P_0[i]$ of plaintexts ($i \in \{0,\ldots,31\}$) with $|P_0[i]| = 2^8$, such that the corresponding $P_1[i]$ are $\Lambda^1$-sets.
  **4.** For all $2^{128}$ round keys $K^7$:
    **5.** Decipher the 32 sets of ciphertexts $P_7[i]$ to get $P_6[i]$ and $Q_6[i]$.
    **6.** Compute $L^5_{0,3}$, $L^6_{1,3}$, and $L^6_{2,3}$.
    **7.** For all $2^{16}$ combinatios of $L^6_{0,3}$ and $L^6_{3,3}$:
      **8.** $i := 0$; reject := false;
      **9.** while $i \leq 31$ and reject=false:
      begin
        **10.** Compute

$$b[i] := \bigoplus_{A \in Q_5[i]} \mathrm{BO}(A_3, 3, L^6_1, L^5_{0,1}).$$

        **11.** If $b[i] = 0$ then i := i+1
                else reject := true.
      end ($*$ while $*$).
      **12.** If reject=false then stop ($*$ and accept key bytes $*$).

The analysis of the attack is essentially the same as its counterpart for RD-256. The only difference is that the loop in step 7 is iterated $2^{16}$ times instead of $2^{32}$. So the attack needs the time of about

$$2^{32} * 2^{128} * 2^{16} * 1 * 2^8 = 2^{184} \text{ basic operations.}$$

## 7 Final Comments, Summary, and Conclusion

In [3], the authors of Rijndael described the Square-6 attack for RD-128. Extensions of this attack for RD-192 and RD-256 were missing, though. The target of the current paper is to close this gap.

The attacks described in this paper are highly impractical. Considering even such certificational attacks as ours is good scientific practice. And the design of Rijndael was determined "by looking at the maximum number of rounds for which shortcut attacks have been found" [3, Chapter 7.6], allowing an additional margin of security. Any attack which is faster than exhaustive search counts as "shortcut attack".

| Attack | target | # Rounds | # Chosen Plaintexts | Time [# basic operations] | Memory [# Ciphertexts] |
|---|---|---|---|---|---|
| simple Square | generic | 4 | $2^8$ | small | small |
| ext. at the end | generic | 5 | $5 * 2^8$ | $2^{48}$ | small |
| Square-6 | generic | 6 | $2^{32}$ | $2^{80}$ | $2^{32}$ |
| 7-round | generic | 7 | $2^{32}$ | $2^{208}$ | $2^{32}$ |
| | RD-192 | 7 | $2^{32}$ | $2^{184}$ | $2^{32}$ |
| | RD-256 | 7 | $2^{32}$ | $2^{200}$ | $2^{32}$ |

**Table 2.** Summary of Results.

Table 2 summarises how the different attacks perform. The results for 4–6 rounds of Rijndael originate from [3]. Note that [3] counted the number of "cipher executions" to measure the running time.

In [5], Fergusen and others describe improved attacks on Rijndael. A preliminary version of [5] has been sent to the current author by one of the authors of [5], and the following remarks are base on that version. [5] describes some weaknesses of the Rijndael key schedule, but does not exploit these for actual attacks. The attacks in [5] are mainly based on improvements of the Square-6 attack, using the "partial sums". E.g., an attack on seven rounds of Rijndael is proposed, which requires $2^{32}$ chosen plaintexts and the time equivalent to $2^{170}$ trial encryptions. Using the observations made in the current paper, this attack can be improved by a factor of $2^{16}$, i.e., only needs the equivalent of $2^{156}$ trial encryptions, instead of $2^{170}$.

Our results exhibit a weakness in the Rijndael key schedule. If, e.g., the words $W[\cdot]$ of the expanded key were generated pseudorandomly using a cryptographically secure pseudorandom bit generator, dedicated attacks could not be more efficient than their generic counterparts.

This does not indicate the necessity to modify the Rijndael key schedule, though. The improvements on the generic case are quite small. If we concentrate on counting the number of rounds for which shortcut attacks exist, the cryptanalytic gain of this paper is one round for RD-192, not more. The authors of Rijndael seem to have anticipated such cryptanalytic results by specifying a high security margin for the number of rounds (two additional rounds for RD-192, compared to RD-128 with its ten rounds).

## References


1. C. D'Halluin, G. Bijnens, V. Rijmen, B., Preneel: "Attack on six round of Crypton", Fast Software Encryption 1999, Springer LNCS 1636, pp. 46–59.
2. J. Daemen, L. Knudsen, V. Rijmen: "The block cipher Square", Fast Software Encryption 1997, Springer LNCS 1267, pp. 149–165.
3. J. Daemen, V. Rijmen: "AES proposal: Rijndael" (2nd version), AES submission.
4. J. Daemen, V. Rijmen: "The block cipher BKSQ", Cardis 1998, Springer LNCS, to appear.
5. N. Ferguson, J. Kelsey, B. Schneier, M. Stay, D. Wagner, D. Whiting: "Improved Cryptanalysis of Rijndael", Fast Software Encryption 2000, Springer LNCS, to appear.
6. C. H. Lim: "Crypton: a new 128-bit block cipher", AES submission.
7. C. H. Lim: "A revised version of Crypton – Crypton V1.0 – ", Fast Software Encryption 1999, Springer LNCS 1636, pp. 31–45.

# A collision attack on 7 rounds of Rijndael

Henri Gilbert and Marine Minier

France Télécom R & D
38-40, rue du Général Leclerc
92794 Issy les Moulineaux Cedex 9 - France
email : henri.gilbert@cnet.francetelecom.fr

**Abstract**

Rijndael is one of the five candidate blockciphers selected by NIST for the final phase of the AES selection process. The best attack of Rijndael so far is due to the algorithm designers ; this attack is based upon the existence of an efficient distinguisher between 3 Rijndael inner rounds and a random permutation, and it is limited to 6 rounds for each of the three possible values of the keysize parameter (128 bits, 196 bits and 256 bits). In this paper, we construct an efficient distinguisher between 4 inner rounds of Rijndael and a random permutation of the blocks space, by exploiting the existence of collisions between some partial functions induced by the cipher. We present an attack based upon this 4-rounds distinguisher that requires $2^{32}$ chosen plaintexts and is applicable to up to 7-rounds for the 196 keybits and 256 keybits version of Rijndael.
Since the minimal number of rounds in the Rijndael parameter settings proposed for AES is 10, our attack does not endanger the security of the cipher, indicate any flaw in the design or prove any inadequacy in selection of number of rounds. The only claim we make is that our results represent improvements of the previously known cryptanalytic results on Rijndael.

## 1 Introduction

Rijndael [DaRi98], a blockcipher designed by Vincent Rijmen and Joan Daemen, is one of the 5 finalists selected by NIST in the Advanced Encryption Standard competition [AES99]. It is a variant of the Square blockcipher, due to the same authors [DaKnRi97]. It has a variable block length $b$ and a variable key length $k$, which can be set to 128, 192 or 256 bits. The recommended $nr$ number of rounds is determined by $b$ and $k$, and varies between 10 and 14. In the sequel we will sometimes use the notation Rijndael/$b$/$k$/$nr$ to refer to the Rijndael variant determined by a particular choice of the $b$, $k$ and $nr$ parameters.

The best Rijndael attack published so far is due to the algorithm designers [DaRi98]. It is a variant of a the "Square" attack, and exploits the byte-oriented structure of Rijndael [DaKnRi97]. This attack is based upon an efficient distinguisher between 3 Rijndael inner rounds and a random permutation. It is stated in [DaRi98] that "for the different block lengths of Rijndael no extensions to 7 rounds faster than exhaustive search have been found".
In this paper we describe an efficient distinguisher between 4 Rijndael inner rounds and a random permutation, and we present resulting 7-rounds attacks of Rijndael/$b$=128 which are substantially faster than an exhaustive key search for the $k = 196$ bits and $k = 256$ bits versions and marginally faster than an exhaustive key search for the $k = 128$ bits version.

This paper is organised as follows. Section 2 provides an outline of the cipher. Section 3 investigates partial functions induced by the cipher and the existence of collisions between such partial functions, and describes a resulting distinguisher for 4 inner rounds. Section 4 presents 7-rounds attacks based on the 4-rounds distinguisher of Section 3. Section 5 concludes the paper.

## 2 An outline of Rijndael/$b = 128$

In this Section we briefly described the Rijndael algorithm. We restrict our description to the $b$=128 bits blocksize and will consider no other blocksize in the rest of this paper.
Rijndael/$b/k/nr$ consists of a key schedule and an iterated encryption function with $nr$ rounds. The key schedule derives $nr + 1$ 128-bit round keys $K_0$ to $K_{nr}$ from the $k = 128, 196$ or 256 bits long Rijndael key $K$. Since attacks presented in the sequel do not use the details of the dependence between round keys, we do not provide a description of the key schedule.
The Rijndael encryption function is the composition of $nr$ block transformations. The current 128-bit block value $B$ is represented by a $4 \times 4$ matrix :

$$B = \begin{array}{|c|c|c|c|} \hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array}$$

The definition of the round functions involves four elementary mappings :

- the $\sigma$=ByteSub byte substitution transforms each of the 16 input bytes under a fixed byte permutation $P$ (the Rijndael S-box).
- the $\rho$=ShiftRow rows shift circularly shifts row $i$ ($i = 0$ to 3) in the $B$ matrix by $i$ bytes to the right.
- the $\mu$=MixColumn is a matrix multiplication by a fixed $4 \times 4$ matrix of non-zero GF($2^8$) elements.
- the $\kappa_r$=KeyAddition is a bitwise addition with a 128-bit round key $K_r$.

The Rijndael cipher is composed by an initial round key addition $\kappa_0$, $nr-1$ inner rounds and a final transformation. The $r$th inner round ($1 \leq r \leq nr-1$) is defined as the $\kappa_r \circ \mu \circ \rho \circ \sigma$ function. The final transformation at the round $nr$ is an inner round without MixColumn mapping : FinalRound $= \kappa_{nr} \circ \rho \circ \sigma$. We can thus summarise the cipher as follows:

$B$:=$\kappa_0(B)$;
For $r = 1$ to $nr-1$
    $B$:=InnerRound($B$);
FinalRound($B$);

**Remarks** :

- $\sigma$ is the single non $GF(8)$-linear function of the whole cipher.
- The Rijndael S-box $P$ is the composition of the multiplicative inverse function in GF(8) (NB : '00' is mapped into itself) and a fixed GF(2)-affine byte transformation. If the affine part of $P$ was omitted, algebraic methods (e.g. interpolation attacks) could probably be considered for the cryptanalysis of Rijndael.
- The $\mu \circ \rho$ linear part of Rijndael appears to have been carefully designed. It achieves a full diffusion after 2 rounds, and the Maximum Distance Separability (MDS) property of $\mu$ prevents good differential or linear "characteristics" since it ensures that two consecutive rounds involve many active S-boxes.

# 3 Distinguishing 4 inner rounds of Rijndael/$b$=128 from a random permutation

## 3.1 Notation

Figure 1 represents 4 consecutive inner round functions of Rijndael associated with any 4 fixed unknown 128-round keys. $Y, Z, R, S$ represent the input blocks of the 4 rounds and $T$ represents the output of the 4th round. We introduce short notations for some particular bytes of $Y, Z, R, S, T$, which play a particular role in the sequel : $y = Y_{0,0}$, $z_0 = Z_{0,0}$, $z_1 = Z_{1,0}$, $z_2 = Z_{2,0}$, $z_3 = Z_{3,0}$, and so on. Finally we denote by $c$ the ($c_0 = Y_{1,0}$, $c_1 = Y_{2,0}$, $c_2 = Y_{3,0}$) triplet of $Y$ bytes.

Let us fix all the $Y$ bytes but $y$ to any 11-uple of constant values. So the $c$ triplet is assumed to be equal to a constant $c = (c_0, c_1, c_2)$ triplet, and the 12 $Y_{i,j}$, i=1 to 3, j=0 to 3 are also assumed to be constant. The $Z, R, S, T$ bytes $z_0$ to $z_3$, $r_0$ to $r_3$, $s$, and $t_0$ to $t_3$ introduced in Figure 1 can be seen as c-dependent functions of the $y$ input byte. In the sequel we sometimes denote by $z_0^c[y]$ to $z_3^c[y]$, $r_0^c[y]$ to $r_3^c[y]$, $s^c[y]$, $t_0^c[y]$ to $t_3^c[y]$ the $z_i$, $r_i$, $s$, $t_i$ byte value associated with a $c$ constant and one $y \in 0..255$ value.

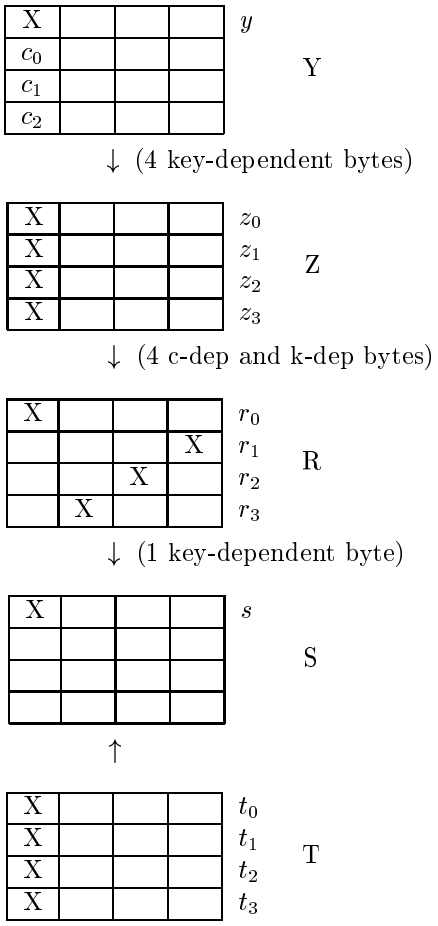



Figure 1: 4 inner rounds of Rijndael

### 3.2 The 3-rounds distinguisher used in the Rijndael/b=128 designers' attack

The Rijndael designers' attack is based upon the observations that :
- bytes $z_0$ to $z_3$ are one to one functions of $y$ and the other $Z$ bytes are constant.
- bytes $r_0$ to $r_3$ are one to one functions of $y$ (as well as the 12 other $R$ bytes).
- $s$ is the XOR of four one to one functions of $y$ and thus $\sum_{y=0}^{255} s[y] = 0$.

Thus 3 consecutive inner rounds of Rijndael have the distinguishing property that if all $Y$ bytes but $y$ are fixed and $y$ is taken equal to each of the 256 possible values, then the sum of the 256 resulting $s$ values is equal to zero.

This leads to a 6-rounds attack (initial key addition followed by 5 inner rounds followed by final round). As a matter of fact an initial round (i.e. an initial key addition followed by 1 inner round) can be added on top, at the expense of testing assumptions on 4 key bytes of the initial key addition. Moreover, two additional rounds can be added at the end (namely one additional inner round followed by one final round), at the expense of testing assumptions on 4 final round key bytes. Combining both extensions provides an attack which requires $2^{32}$ plaintexts and has a complexity of $2^{72}$ encryptions.

### 3.3 A 4-rounds distinguisher for Rijndael/$b = 128$

We now analyse in detail the dependency of the byte oriented functions introduced in Section 3.1 in the $c$ constant and the expanded key. We show that the $s^c[y]$ function is entirely determined by a surprisingly small number of unknown bytes, which either only depend upon the key or depend upon both the key and the $c$ value, and that as a consequence there exist $(c', c'')$ pairs of distinct $c$ values such that the $s^{c'}[\cdot]$ and $s^{c''}[\cdot]$ partial functions collide, i.e. $s^{c'}[y] = s^{c''}[y]$ for $y = 0, 1, \cdots, 255$. This provides an efficient test for distinguishing 4 inner rounds of Rijndael from a random permutation.

The construction of the proposed distinguisher is based upon the following observations, which are illustrated in Figure 1.

**Property 1 :** At round 1, the $y \rightarrow z_0^c[y]$ one to one function is independent of the value of the $c$ triplet and is entirely determined by one key byte. The same property holds for $z_1, z_2, z_3$. This is because at the output of the first round ShiftRow the $c_0$ to $c_2$ constants only affect columns 1 to 3 of the current block value, whereas the $z_0$ to $z_3$ bytes entirely depend upon column 0. For similar reasons, the other bytes of $Z$ are independent of $y$ : each of the bytes of column 1 (resp 2, resp 3) of $Z$ is entirely determined by the $c_0$ (resp $c_1$, resp $c_2$) byte and one key-dependent byte.
More formally, there exist 16 MixColumn matrix coefficients $a_{i,j}$,$i$=0..3, $j$=0..3 and 16 key-dependent constants $b_{i,j}$, $i$=0..3, $j$=0..3 such that $z_i = a_{i,0}P(y) + b_{i,0}$, $i$=0..3 and $z_{i,j} = a_{i,0}P(c_{j-1}) + b_{i,j}$, $i$=1..3, $j$=0..3.

**Property 2 :** At round 2, each of the four bytes $r_i[y]$, $i = 0..3$ is a one to one function of $z_i[y]$, and the $r_i[y] \rightarrow z_i[y]$ is entirely determined by one single unknown constant byte that is entirely determined by $c$ and the key.

More formally, there exist 16 MixColumn coefficients $\alpha_i$, $i = 0..3$, $\beta_i$, $i = 0..3$ $\gamma_i$, $i = 0..3$ and $\delta_i$, $i = 0..3$ and 4 key-dependent constants $\epsilon_i$, $i = 0..3$ such that $r_i = \alpha_i \cdot P(z_{i,0}) + \beta_i \cdot P(z_{i,1}) + \gamma_i \cdot P(z_{i,2}) + \delta_i \cdot P(z_{i,3}) + \epsilon_i$, $i = 0..3$. The $r_i$ bytes are thus related to $c$ and $y$ by the relations : $r_i = \alpha_i \cdot P(a_{i,0}P(y) + b_{i,0}) + \beta_i \cdot P(a_{i,1}P(c_0) + b_{i,1}) + \gamma_i \cdot P(a_{i,2}P(c_1) + b_{i,2}) + \delta_i \cdot P(a_{i,3}P(c_2) + b_{i,3}) + \epsilon_i$, $i = 0..3$.
Consequently, the $r_0[y]$ to $r_3[y]$ one to one functions of $y$ are entirely determined by the 4 key-dependent constant unknown bytes $b_{i,0}$ introduced in property (1) and the 4 $c$- and $k$-dependent bytes $b_i = \beta_i \cdot P(a_{i,1}P(c_0) + b_{i,1}) + \gamma_i \cdot P(a_{i,2}P(c_1) + b_{i,2}) + \delta_i \cdot P(a_{i,3}P(c_2) + b_{i,3}) + \epsilon_i$, i$= 0 \cdot \cdot 3$.

**Property 3 :** At round 3, the $s$ byte can be expressed as a function of the $r_0$ to $r_3$ bytes and one $c$-independent and key-dependent unknown constant. Consequently, the $s^c[y]$ function is entirely determined by 4 key-dependent and $c$-dependent constants and 5 $c$-independent and key-dependent constants.

**Property 4 :** Let us consider the decryption of the fourth inner round : $s$ can be expressed as $s = p^{-1}[(0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3) + k_5]$ where $p$ represents the single S-box. In other words $0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3$ is a one to one function of $s$, and that function is entirely determined by one single key byte $k_5$. Thus $0E.t_0 + 0B.t_1 + 0D.t_2 + 09.t_3$ is a function of $y$ that is entirely determined by 6 unknown bytes which only depend upon the key and by 4 additional unknown bytes which depend both upon $c$ and the key.

The above properties provide an efficient 4-rounds distinguisher. We can restate property (3) in saying that the $s^c[y]$ function is entirely determined (in a key-dependent manner) by the 4 $c$-dependent bytes $b_0$ to $b_3$. Let us make the heuristic assumption that these 4 unknown $c$-dependent bytes behave as a random function of the $c$ triplet of bytes. By the birthday paradox, given a $C$ set of about $2^{16}$ $c$ triplet values, there exist with a non negligible probability two distinct $c'$ and $c''$ in $C$ such that the $s^{c'}[y]$ and $s^{c''}[y]$ functions induced by $c'$ and $c''$ are equal (i.e. in other words such that the $(s^{c'}[y])_{y=0..255}$ and $(s^{c''}[y])_{y=0..255}$ lists of 256 bytes are equal). Property (4) provides a method to test such a "collision", using the $t_0$ to $t_3$ output bytes of 4 inner rounds : $c'$ and $c''$ collide if and only if $\forall y \in [0, ..., 255]$, $0E.t_0^{c'} + 0B.t_1^{c'} + 0D.t_2^{c'} + 09.t_3^{c'} = 0E.t_0^{c''} + 0B.t_1^{c''} + 0D.t_2^{c''} + 09.t_3^{c''}$. Note that it is sufficient to test the above equality on a limited number of $y$ values (say 16 for instance) to know with a quite negligible "false alarms" probability whether the $s^{c'}[y]$ and $s^{c''}[y]$ functions collide.

We performed some computer experiments which confirmed the existence, for arbitrarily chosen key values, of $(c', c'')$ pairs of $c$ value such that the $s^{c'}[y]$ and $s^{c''}[y]$ functions collide. For some key values, we could even find four byte values $c'_1$, $c'_2$, $c''_1$ and $c''_2$ such that for each of the 256 possible values of the

$c_0$ byte, the $s^{c'}[y]$ and $s^{c''}[y]$ functions associated with the $c' = (c_0, c'_1, c'_2)$ and $c'' = (c_0, c''_1, c''_2)$ triplets of bytes collide. This stronger property, which is rather easy to explain using the expression of the $b_i$ constants introduced in Property (2), is not used in the sequel.

The proposed 4 rounds distinguisher uses the collision test derived from property (4) is the following manner :

- select a $C$ set of about $2^{16}$ $c$ triplet values and a subset of $\{0..255\}$, say for instance a $\Lambda$ subset of 16 $y$ values.
- for each $c$ triplet value, compute the $L_c = (0E.t_0^c + 0B.t_1^c + 0D.t_2^c + 09.t_3^c)_{y \in \Lambda}$. We claim that such a computation of 16 linear combinations of the outputs represents substantially less than one single Rijndael operation.
- check whether two of the above lists, $L_{c'}$ and $L_{c''}$ are equal. The 4 round distinguisher requires about $2^{20}$ chosen inputs $Y$, and since the collision detection computations (based on the analysis of the corresponding $T$ values) require less operations than the $2^{20}$ 4-inner rounds computations, the complexity of the distinguisher is less than $2^{20}$ Rijndael encryptions.

Note that property (4) also provides another method to distinguish 4 inner round from a random permutation, using $N \leq 256$ plaintexts and $2^{80}$ $N$ operations, namely performing an exhaustive search of the 10 unknown constants considered in property (4). Note that a value such as $N = 16$ is far sufficient in practice. However, we only consider in the sequel the above described birthday test, which provides a more efficient distinguisher.

# 4 An attack of the 7-rounds Rijndael/$b$=128 cipher with $2^{32}$ chosen plaintexts

In this Section we show that the 4 inner rounds distinguisher of Section 3 provides attacks of the 7-rounds Rijndael for the $b$=128 blocksize and the various keysizes. We present two slightly different attacks. The first one (cf Section 4.2 hereafter) is substantially faster than an exhaustive search for the $k$=196 and $k$=256 keysizes, but slower than exhaustive search for the $k$=128 bits keysize. The second attack (cf Section 4.2) is dedicated to the $k$=128 keysize, and is marginally faster than an exhaustive search for that keysize.

The 7-rounds Rijndael is depicted at Figure 2. $X$ represents a plaintext block, and $V$ represents a ciphertext block. In Figure 2 the 4 inner rounds of Figure 1 are surrounded by one initial $X \rightarrow Y$ round (which consists of an initial key addition followed by one round), and two final rounds (which consist of one $T \rightarrow U$ inner round followed by an $U \rightarrow V$ final round).

Our attack method is basically a combination of the 4-round distinguisher presented in Section 3 and an exhaustive search of some keybytes (or combinations of keybytes) of the initial and the two final rounds. In the attack of Section

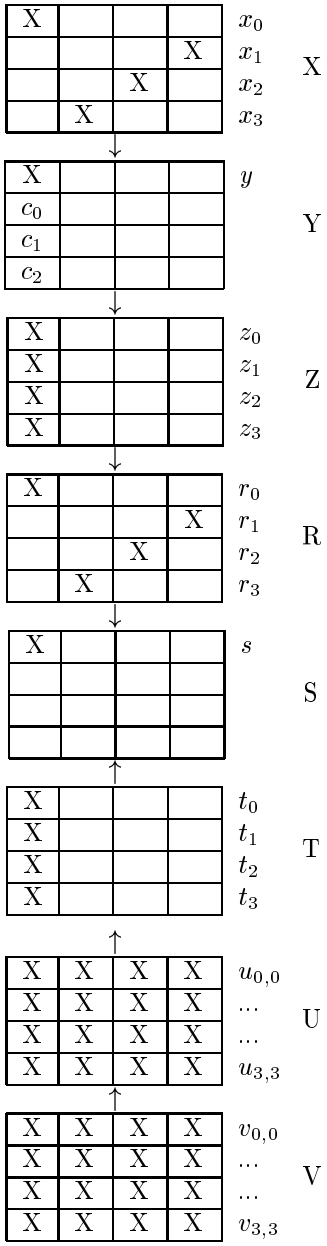



Figure 2: 7-rounds Rijndael

4.1 we are using the property that in the equations provided by the 4-rounds distinguisher there is a variables separations in terms which involve one half of the 2 last rounds key bytes and terms which involve a second half of the 2 last round key bytes in order to save a $2^{80}$ factor in the exhaustive search complexity. In the attack of Section 4.2, we are using precomputations on colliding pairs of $c$ values to test each 128-bits key assumption with less operations than one single Rijndael encryption.

### 4.1 An attack of the 7-rounds Rijndael/$b$=128/$k$=196 or 256 with $2^{32}$ chosen plaintexts and a complexity of about $2^{140}$

We now explain the attack procedure in some details, using the notation introduced in Figure 2. We fix all $X$ bytes except the four bytes $x_0$ to $x_3$ equal to 12 arbitrary constant values. We encrypt the $2^{32}$ plaintexts obtained by taking all possible values for the $x_0$ to $x_3$ bytes, thus obtaining $2^{32}$ $V$ ciphertext blocks. We are using the two following observations :

**Property 5 :** If the 4 key bytes added with the $x_0$ to $x_3$ bytes in the initial key addition are known (let us denote them by $k_{ini} = (k_0, k_1, k_2, k_3)$, then it is possible to partition the $2^{32}$ plaintexts in $2^{24}$ subsets of 256 plaintext values satisfying the conditions of Section 3, i.e. such that the corresponding 256 $Y$ values satisfy the following conditions :
- the y byte takes 256 distinct values (which are known up to an unknown constant first round key byte which is not required for the attack).
- the $c = (c_0, c_1, c_2)$ triplet of bytes is constant ; moreover, each of the $2^{24}$ subsets corresponds to a distinct $c$ value (the $c$ value corresponding to each subset is known up to three constant first round keybytes which are not required for the attack).
- the 12 other $Y$ bytes are constant and their constant values $Y_{i,j}$ for $i$=1..3 and $j$=0..3 is the same for all subsets.
Note that the same property is used in the Rijndael designers' attack.

**Property 6 :** Each of the $t_0, t_1, t_2, t_3$ bytes can be expressed as a function of four bytes of the $V$ ciphertext and five unknown key bytes (i.e. 4 of the final round key bytes and one linear combination of the penultimate round key bytes). Therefore, we can "split" the $t^c[y] =' 0E'.t_0^c[y] +' 0B'.t_1^c[y] +' 0D'.t_2^c[y] +' 09'.t_3^c[y]$ combination of the four $t_i^c[y]$ bytes considered in the 4-rounds distinguisher as the XOR of two terms $\tau_1^c[y]$ and $\tau_2^c[y]$ which can both be expressed as a function of 8 ciphertext bytes and 10 unknown key bytes, namely $\tau_1 =' 0E'.t_0^c[y] +' 0B'.t_1^c[y]$ and $\tau_2 =' 0D'.t_2^c[y] +' 09'.t_3^c[y]$. We denote in the sequel by $k_{\tau_1}$ those 10 unknown keybytes which allow to derive $\tau_1$ from 8 bytes of the $V$ ciphertext, and by $k_{\tau_2}$ those 10 keybytes which allow to derive $\tau_1$ from 8 bytes of the $V$ ciphertext.

We perform an efficient exhaustive search of the $k_{ini}$, $k_{\tau_1}$ and $k_{\tau_2}$ keys in the following way :

- For each of the $2^{32}$ possible $k_{ini}$ assumptions, we can partition the set of the $256^4$ possible $X$ values in $256^3$ subsets of 256 $X$ values each, according to the value of the $c$ constant, and select say $256^2$ of these $256^3$ subsets. Thus each of the $256^2$ selected subsets is associated with a distinct value of the $c$ constant. Note that the $c$ value associated with a subset and the $y$ values associated with each of the $X$ plaintexts of a subset are only known up to unknown keybits, but this does not matter for our attack. We can denote by $c*$ and $y*$ the known values which only differ from the actual values by fixed unknown key bits.

- Now for each subset associated with a $c*$ constant triplet, based on the say 16 ciphertexts associated with the $y* = 0$ to $y* = 15$ values, we can precompute the $(\tau_1^c(y))_{y*=0..15}$ 16-tuple of bytes for each of the $2^{80}$ possible $k_{\tau_1}$ keys. We can also precompute the $(\tau_2^c(y))_{y*=0..15}$ 16-tuple for each of the $2^{80}$ possible $k_{\tau_2}$ keys.

Based on this precomputation, for each $(c'*, c''*)$ pair of distinct c* values :

- We precompute a (sorted) table the $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15}$ 16-tuple of bytes for each of the $2^{80}$ possible $k_{\tau_1}$ keys (the computation of each 16-tuple just consists in xoring two precomputed values)

- For each of the $2^{80}$ possible values of the $k_{\tau_2}$ key, we compute the $(\tau_2^{c'}(y) \oplus \tau_2^{c''}(y))_{y*=0..15}$ 16-tuple of bytes associated with the observed ciphertext, and check whether this t-uple belongs to the precomputed table of 16-tuple $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15}$. If for a given $k_{\tau_1}$ value there exists a $k_{\tau_2}$ value such that $(\tau_1^{c'}(y) \oplus \tau_1^{c''}(y))_{y*=0..15} = (\tau_2^{c'}(y) \oplus \tau_2^{c''}(y))_{y*=0..15}$, (i.e. $t^{c'}[y] = t^{c''}[y]$ for each of the $y$ values associated with $y* = 0$ to 16, this represents an alarm). The equality between the t bytes associated with $c'$ and $c''$ is checked for the other $y*$ values. If the equality is confirmed, this means that a collision between the $s^c[y]$ functions associated with $c'$ and $c''$. This provides 20 bytes of information on the last and penultimate key values, since with overwhelming probability, the right values of $k_{ini}$, $k_{\tau_1}$ and $k_{\tau_2}$ have then been found.

Since the above procedure tests whether the exist collisions inside a random set of $256^2$ of the $256^4$ possible $s^c[y]$ functions, the probability of the procedure to result in a collision, and thus to provide $k_{ini}$, $k_{\tau_1}$ and $k_{\tau_2}$ is high (say about 1/2). In other words, the success probability of the attack is about 1/2.

Once $k_{ini}$, $k_{\tau_1}$ and $k_{\tau_2}$ have been found, the 16-bytes final round key is entirely determined and the final round can be decrypted, so one is left with the problem of cryptanalysing the 6-round version of Rijndael. One might object that the last round of the left 6-round cipher is not a final round, but an inner round. However, it is easy to see that by applying a linear one to one change of variable to $U$ and the 6th round key (i.e. replacing $U$ by a $U'$ linear function of $U$ and $K_6$ by a linear function $K'_6$ of $K_6$), the last round can be represented as a final round (i.e. $U'$ is the image of $T$ by the final round associated with

$K'_6$). Thus we are in fact left with the cryptanalysis of the 6-round Rijndael, and the last round subkey is easy to derive. The process of deriving the subkeys of the various rounds can then be continued (using a subset of the $2^{32}$ chosen plaintexts used for the derivation of $k_{ini}$, $k_{\tau_1}$ and $k_{\tau_2}$), with negligible additional complexity, until the entire key has eventually been recovered.

### 4.2 An attack of the 7-rounds Rijndael/$b$=128/$k$=128 $2^{32}$ chosen plaintext

We now outline a variant of the former attack that is dedicated to the $k$=128 bits version of Rijndael and is marginally faster than an exhaustive search. This attack requires a large amount of precomputations.

As a matter of fact, it can be shown that the 4 $c$-dependent bytes that determine the mapping between four $z_i^c[y]$ bytes and the four $r_i^c[y]$ are entirely determined by 12 key-dependent (and c-independent) bytes. For each of the $256^{12}$ possible values of this $\phi(K)$ 12-tuple of bytes, we can compute colliding $c'$ and $c''$ triplets of bytes (this can be done performing about $256^2$ partial Rijndael computations corresponding to say $256^2$ distinct $c$ values and looking for a collision. One can accept that no collision be found for some $\phi(K)$ values : this just means that the attack will fail for a certain fraction (say 1/2) of the key values. We store $c'$ and $c''$ (if any) in a table of $256^{12}$ $\phi(K)$ entries.

Now we perform an exhaustive search of the $K$ key. To test a key assumption, we compute the $k_{ini}$, $k_{\tau_1}$, $k_{\tau_2}$ and $\phi(K)$ values. Then we find the $(c', c'')$ pair of colliding $c$ values in the precomputed table, compute the two associated $c'*$ and $c''$ values, determine which two precomputed lists $(V^{c'}[y])_{y*=0..15}$ $(V^{c''}[y])_{y*=0..15}$of 16 ciphertext values each are associated with $c'*$ and $c''*$, and finally compute the associated $(t^{c'}[y])_{y*=0..15}$ and $(t^{c''}[y])_{y*=0..15}$ bytes by means of partial Rijndael decryption. The two values associated with $y* = 0$ are first computed and compared. The two values associated with $y* = 1$ are only computed and compared if they are equal, etc, thus in average only two partial decryption are performed. It the two lists of 16 $t$ bytes are equal, then there is an alarm, and the $K$ is further tested with a few plaintexts and ciphertexts.

We claim than the complexity of the operations performed to test one $K$ key is marginally less than one Rijndael encryption.

## 5 Conclusion

We have shown that the existence of collisions between some partial byte oriented functions induced by the Rijndael cipher provides a distinguisher between 4 inner rounds of Rijndael and a random permutation, which in turn enables to mount attacks on a 7-rounds version of Rijndael for any key-length.
Unlike many recent attacks on block ciphers, our attacks are not statistical in

nature. They exploit (using the birthday paradox) a new kind of cryptanalytic bottleneck, namely the fact that a partial function induced by the cipher (the $s^c[y]$ function) is entirely determined by a remarkably small number of unknown constants. Therefore, unlike most statistical attacks, they require a rather limited number of plaintexts (about $2^{32}$). However, they are not practical because of their high computational complexity, and do not endanger the full version of Rijndael. Thus we do not consider they represent arguments against Rijndael in the AES competition.

# References


[AES99] http://www.nist.gov/aes

[DaKnRi97] J. Daemen, L.R. Knudsen and V. Rijmen, "The Block Cipher Square". In *Fast Software Encryption - FSE'97* , pp. 149-165, Springer Verlag, Haifa, Israel, January 1997.

[DaRi98] J. Daemen, V. Rijmen, "AES Proposal : Rijndael", *The First Advanced Encryption Standard Candidate Conference,* N.I.S.T., 1998.

# Relationships among Differential, Truncated Differential, Impossible Differential Cryptanalyses against Word-Oriented Block Ciphers like Rijndael, E2

Makoto Sugita[1], Kazukuni Kobara[2], Kazuhiro Uehara[1], Shuji Kubota[1], Hideki Imai[2]
[1] NTT Wireless Systems Innovation Laboratory, Network Innovation Laboratories,
1-1 Hikari-no-oka, Yokosuka-shi, Kanagawa, 239-0847 Japan
E-mail: {sugita, uehara, kubota}@wslab.ntt.co.jp
[2] Institute of Industrial Sciences, The University of Tokyo
Roppongi, Minato-ku, Tokyo 106-8558, Japan
E-mail:{kobara, imai}@imailab.iis.u-tokyo.ac.jp

**Abstract**

We propose a new method for evaluating the security of block ciphers against differential cryptanalysis and propose new structures for block ciphers. To this end, we define the word-wise Markov (Feistel) cipher and random output-differential (Feistel) cipher and clarify the relations among the differential, the truncated differential and the impossible differential cryptanalyses of the random output-differential (Feistel) cipher. This random output-differential (Feistel) cipher model uses a not too strong assumption because denying this approximation model is equivalent to denying truncated differential cryptanalysis. Utilizing these relations, we evaluate the truncated differential probability and the maximum average of differential probability of the word-wise Markov (Feistel) ciphers like Rijndael, E2 and the modified version of block cipher E2. This evaluation indicates that all three are provably secure against differential cryptanalysis, and that Rijndael and a modified version of block cipher E2 have stronger security than E2.

**keywords.** truncated differential cryptanalysis, truncated differential probability, maximum average of differential probability, generalized E2-like transformation, SPN-structure, word-wise Markov cipher, random output-differential cipher

## 1 Introduction

As a measure of the security of block ciphers, the maximum average of differential probability was defined by Nyberg and Knudsen [15] by generalizing provable security against differential cryptanalysis as introduced by Biham and Shamir [2]. Based on this idea, many new block ciphers have been proposed, e.g. the block cipher MISTY was proposed by M. Matsui [10]. It was designed on the basis of the theory of provable security against differential and linear cryptanalysis.

The block cipher E2 was proposed in [6] as an AES candidate. This cipher uses Feistel structures as a global structure like DES, and uses the SPN (Substitution and Permutation Network)-structure in its S-boxes. [6] said this cipher can be 'proved' to offer immunity against differential cryptanalysis by counting the maximum number of active S-boxes. However, Sugita proposed a method for evaluating the maximux average of differential probability of SPN-structures, and then evaluated the SPN-structure of E2[16, 17]. Using the similar method, Matsui stated that 8-rounds E2 can be defeated by truncated differential cryptanalysis [19, 14], which implies that just counting the maximum number of active S-boxes is not sufficient for proving the security of block ciphers.

The block cipher Rijndael was also proposed as an AES candidate [3]. This cipher uses the SPN (Substitution and Permutation Network)-structure as its basic structure. The basis for proving its security against differential cryptanalysis involves a similar evaluation method as used for E2. Therefore more accurate proof is necessary.

In this paper, we introduce the word-wise Markov (Feistel) cipher and random output-differential (Feistel) cipher as a approximation model for the accurate definition of truncated differential probability, and clarify the relationships among differential, truncated differential and impossible differential cryptanalyses, and propose a new method for evaluating the security of block ciphers against differential, truncated differential and impossible differential cryptanalysis under this model, and propose new structures for block ciphers that are secure against these cryptanalyses. This random output-differential (Feistel) cipher model does not use too strong an assumption because denying this model is equivalent to denying truncated differential cryptanalysis.

This report is organized as follows.

Section 2 defines the structures of word-oriented block ciphers like SPN-Structures, PSN-structures and the E2($'$)-like transformation.

Section 3 defines the differential probability, and defines the word-wise Markov (Feistel) cipher, random output-differential (Feistel) cipher, and using these definitions, defines the truncated differential probability.

Section 4 clarifies the relations between the truncated differential probability and the differential probability of the random output-differential (Feistel) cipher. It then describes a procedure for calculating the truncated differential probability and (maximum average of) the differential probability of typical random output-differential ciphers like SPN-structures including Rijndael and E2($'$)-like transformations. It proves that both Rijndael and the modified E2-like transformation are provably secure against differential, truncated differential and impossible differential cryptanalysis if they can be approximated as random output-differential (Feistel) ciphers.

Section 5 concludes this paper.

# 2 Structures of Word-oriented Block Ciphers

## 2.1 Word-oriented Block Ciphers

A word-oriented block cipher is a block cipher whose input and output data is a set of input words of fixed size, and whose operations consist only of word-wise operations of fixed size. In the usual case, the word size is 8, i.e. byte size. Example of these ciphers include Rijndael, E2, etc.

## 2.2 Feistel Structures

Associate with a function $f : GF(2)^n \to GF(2)^n$, a function $\delta_{2n,f}(L, R) = (R \oplus f(L), L)$ for all $L, R \in GF(2)^n$. $\delta_{2n,f}$ is called the Feistel transformation associated with $f$. Furthermore, for functions $f_1, f_2, \cdots, f_s : GF(2)^n \to GF(2)^n$, define $\psi_n(f_1, f_2, \cdots, f_s) = \delta_{2n,f_s} \circ \cdots \circ \delta_{2n,f_2} \circ \delta_{2n,f_1}$. We call $D(f_1, f_2, \cdots, f_s) = \psi_n(f_1, f_2, \cdots, f_s)$ as the $s$-round Feistel structure. At this time, we call the functions $f_1, f_2, \cdots, f_s$ as S-boxes of the Feistel structure $D(f_1, f_2, \cdots, f_s)$.

## 2.3 SPN-Structures and PSN-Structures

[11] defines SPN-Structures. First we define the 3-layer SPN-structure.

This structure consists of two kinds of layers, i.e. nonlinear layer and bijective linear layer. Each layer has different features as follows.

**Nonlinear layer**: This layer is composed of $m$ parallel $n$-bit bijective nonlinear transformations.

**Linear layer**: This layer is composed of linear transformations over the field $GF(2^n)$ (especially in the case of E2, bit-wise XOR), where inputs are transformed linearly to outputs per word ($n$-bits).

Next for $s \in \mathbf{N}$ we define the $s$-layer SPN-structure, which consists of $s$ layers. First is a nonlinear layer, second is a linear layer, third is a nonlinear layer, $\cdots$ .

Similarly, for $s \in \mathbf{N}$ we define the $s$-layer PSN-structure. This layer consists of $s$ layers. First is a linear layer, second is a nonlinear layer, third is a linear layer, $\cdots$.

The SPN-structure is the basic structure of Rijndael, a candidate for AES. We will analyze the security of Rijndael afterwards.

### 2.4 E2($'$)-like Transformations

[6] proposed the block cipher E2. This cipher has Feistel structures and its S-boxes are composed of 3-layers SPN structures. We generalize this structure, and define E2-like transformations as Feistel structure with S-boxes composed of $s$-layers (in the case of E2, 3-layers) SPN-structures.

Similarly, we define E2$'$-like transformations as Feistel structures with S-boxes composed of $s$-layer PSN-structures.

## 3 Differential Probability, Truncated Differential Probability, Word-wise Markov (Feistel) Cipher and Random Output-Differential (Feistel) Cipher

This section defines the (maximum average of) differential probability, truncated differential probability, word-wise (Feistel) Markov cipher and random output-differential (Feistel) cipher.

### 3.1 Differential Probability of Block Ciphers

We define the differential of block ciphers. We consider the encryption of a pair of distinct plaintexts by an $r$-round iterated cipher. Here the round function $Y = f(X, Z)$ is such that, for every round sub-key $Z$, $f(\ \cdot\ , Z)$ establishes a one-to-one correspondence between the round input $X$ and the round output $Y$. Let the "difference" $\Delta X$ between two plain-texts (or two cipher texts) $X$ and $X^*$ be defined as

$$\Delta X = X \oplus X^*.$$

From the pair of encryption results, one obtains the sequence of differences $\Delta X(0), \Delta X(1), \cdots, \Delta X(r)$ where $X(0) = X$ and $X(0)^* = X^*$ denote the plaintext pair (such that $\Delta X(0) = \Delta X$) and where $X(i)$ and $X^*(i)$ for $(0 < i < r)$ are the outputs of the $i$-th round, which are also the inputs to the $(i+1)$-th round. The sub-key for the $i$-th round is denoted as $Z^{(i)}$.

Next we define the $i$-th round differential and maximum average of differential probabilities.

**Definition 1** *[7] An $i$-round differential is the couple $(\alpha, \beta)$, where $\alpha$ is the differential of a pair of distinct plaintexts $X$ and $X^*$ and $\beta$ is a possible difference for the resulting $i$-th round outputs $X(i)$ and $X^*(i)$. The probability of an $i$-round differential $(\alpha, \beta)$ is the conditional probability that $\beta$ is the difference, $\Delta X(i)$, of the cipher text pair after $i$ rounds given that the plaintext pair $(X, X^*)$ has difference $\Delta X = \alpha$ when the plaintext $X$ and the sub-keys $Z^{(1)}, \cdots, Z^{(i)}$ are independent and uniformly random. We denote this differential probability by $P(\Delta X(i) = \beta | \Delta X = \alpha)$.*

The probability of an $s$-round differential is known to satisfy the following property.

**Lemma 1** *[7] For the Markov cipher, the probability of an $s$-round differential equals*

$$P(\Delta X(s) = \beta(s) | \Delta X(0) = \beta(0)) = \sum_{\beta(1)} \sum_{\beta(2)} \cdots \sum_{\beta(s-1)} \prod_{i=1}^{s} P(\Delta X(i) = \beta(i) | \Delta X(i-1) = \beta(i-1)).$$

We define the maximum average of differential probability as follows. This value is known to be the best measure with which to confirm that block ciphers are secure against differential cryptanalysis.

**Definition 2** *[15] We define the maximum average of differential probability* $ADP^{(s)}_{\max}$ *by*

$$\mathrm{ADP}^{(s)}_{\max} = \max_{\alpha \neq 0, \beta} P(\Delta X(s) = \beta | \Delta X = \alpha).$$

## 3.2 Word-wise Markov (Feistel) Cipher

[5] uses the truncated differential for the cryptanalysis of word-oriented block ciphers. However, the accurate definition of truncated differential probability is not offered because this cryptanalysis is essentially based on approximation. In this subsection, in order to legitimate this notion, we redefine the truncated differential probability of word-oriented block ciphers.

We consider the encryption of a pair of distinct plaintexts by an $r$-round iterated cipher. Here the round function $Y = f(X, Z)$ is such that, for every round sub-key $Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'}$, $f(\,\cdot\,, Z)$ establishes a one-to-one correspondence between the round input $X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m$ and the round output $Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m$.

We define a characteristic function $\chi : GF(2^n)^m \to GF(2)^m$, $(x_1, \cdots, x_m) \longmapsto (y_1, \cdots, y_m)$ by

$$y_i = \begin{cases} 0 & \text{if } x_i = 0 \\ 1 & \text{otherwise,} \end{cases}$$

Hereafter, we call $\chi(x)$ as a characteristic of $x \in GF(2^n)^m$.

For the definition of the truncated differential probability, we define the word-wise Markov cipher as a real block-cipher model, in the same way as the Markov cipher was in [7]

**Definition 3** *A word-oriented cipher with round function* $Y = f(X, Z) (X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m,\ Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m, Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'})$*, is a word-wise Markov cipher if for all choices of* $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_m) \in GF(2^n)^m (\alpha \neq 0)$*,* $\beta = (\beta_1, \beta_2, \cdots, \beta_m) \in GF(2^n)^m (\beta \neq 0)$ *and* $p \in \{1, 2, \cdots, m'\}$*,*

$$P(\Delta Y_p = \beta_p | \Delta X = \alpha, X = \gamma)$$

*is independent of* $\gamma$*, and* $P(\Delta Y_p = \beta_p | \Delta Y_p \neq 0, \Delta X = \alpha, X = \gamma)$ $(p = 1, 2, \cdots, m)$ *are jointly statistically independent when the sub-key* $Z$ *is uniformly random, or, equivalently, if*

$$P(\Delta Y_p = \beta_p | \Delta X = \alpha, X = \gamma) = P(\Delta Y_p = \beta_p | \Delta X = \alpha)$$

*for all choices of* $\gamma$ *and* $P(\Delta Y_p = \beta_p | \Delta Y_p \neq 0, \Delta X = \alpha)$ $(p = 1, 2, \cdots, m)$ *are jointly statistically independent when the sub-key* $Z$ *is uniformly random, where* $\Delta X = (\Delta X_1, \Delta X_2, \cdots, \Delta X_m)$*,* $\Delta Y = (\Delta Y_1, \Delta Y_2, \cdots, \Delta Y_m)$ *are the differential of* $X$*,* $Y$*, respectively.*

**Example.** the PSN-structure is a word-wise Markov cipher, if every bijective nonlinear function in a nonlinear layer consists of a concatenation of XOR and substitution (like DES does). Therefore, block cipher Rijndael and the S-boxes of block cipher E2 are also word-wise Markov ciphers with the same kind of nonlinear functions.

We expand this definition to the Feistel cipher.

**Definition 4** *We define a word-wise Markov Feistel cipher as a Feistel cipher whose S-boxes are word-wise Markov ciphers.*

**Example.** E2′-like transformation is a word-wise Markov Feistel cipher because the PSN-structure is a word-wise Markov cipher if every nonlinear function in a nonlinear layer consists of the concatenation of XOR of the key and substitution (like DES does).

### 3.3 Random Output-Differential (Feistel) Cipher

As preparation for defining the random output-differential cipher, we define the random output-differential transformation.

**Definition 5** *A word-oriented transformation* $Y = g(X, Z)$ $(X = (X_1, X_2, \cdots, X_m) \in GF(2^n)^m$, $Y = (Y_1, Y_2, \cdots, Y_m) \in GF(2^n)^m$, $Z = (Z_1, Z_2, \cdots, Z_{m'}) \in GF(2^n)^{m'})$, *is a random output-differential transformation, if for any input-differential value* $\alpha$, *the following relation is satisfied,*

$$P(\Delta Y = \beta | \Delta X = \alpha) = p^{\mathrm{h}(\chi(\beta))} P(\chi(\Delta Y) = \chi(\beta) | \Delta X = \alpha),$$

*when keys are randomly selected, where* h *is the function that indicates the Hamming weight of the input value,* $p = 1/(2^n - 1)$, *and* $\Delta X = (\Delta X1, \Delta X_2, \cdots, \Delta X_m)$, $\Delta Y = (\Delta Y_1, \Delta Y_2, \cdots, \Delta Y_m)$ *are the differential of* $X$, $Y$, *respectively.*

Using this definition, we define the random output-differential cipher for word-oriented block cipher as approximation model of word-wise Markov cipher.

**Definition 6** *A word oriented cipher with round functions* $X(i+1) = f(X(i), Z^{(i)})(i = 0, 1, \cdots, r-1)$, *where* $Z^{(i)}(i = 0, 1, \cdots)$ *are sub-keys, is a random output-differential cipher if for any random output-differential transformation* $X(0) = g(X, Z^{(0)})$, *the composite transformation* $X(1) = f(g(X, Z^{(0)}), Z^{(1)})$ *is also a random output-differential transformation.*

*At this time, we call a round function which composes a random output-differential cipher by concatenating, as random output-differential round function.*

We expand this definition to the Feistel cipher.

**Definition 7** *A Feistel cipher with S-boxes* $Y = f(X, Z^{(i)})$ $(i = 0, 1, \cdots)$, *where* $Z^{(i)}(i = 0, 1, \cdots)$ *are sub-keys and i-th round output is* $X(i) = (X(i)_L, X(i)_R)$, *is a random output-differential Feistel cipher, if its S-boxes are random output-differential ciphers and the round function of the Feistel cipher*

$$(X(i+1)_L, X(i+1)_R) = (X(i)_R, X(i)_L \oplus f(X(i)_R, Z^{(i)}))$$

*is a random output-differential round function.*

Matsui stated in his presentation of [14] that 8-round E2 can be cryptanalyzed by truncated differential cryptanalysis only assuming randomness of keys. However, this is not accurate, because he tacitly assumes this random output-differential cipher as an approximation model of E2 in his explanation.

However, this approximation may be effective for word-wise Markov (Feistel) cipher like E2, E2′-like transformation and Rijndael. In fact, in the case of E2′-like transformation with 2-layer PSN-structures, which is also a word-wise Markov Feistel cipher for example, let the $\Delta X \in GF(2^n)^{2m}$ be a input differential of this cipher, if the input-differential of S-box $\Delta W = (\Delta W_1, \Delta W_2, \cdots, \Delta W_m) \in GF(2^n)^m (\chi(\Delta W) = \gamma' \in GF(2)^m)$ is randomly distributed with the probability $P(\Delta W = \gamma | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p^{\mathrm{h}(\gamma')}$ for all $\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_m)$ (where $\chi(\gamma) = \gamma'$), then the output-differential of S-box $\Delta U = (\Delta U_1, \Delta U_2, \cdots, \Delta U_m) \in GF(2^n)^m$ $(\chi(\Delta U) = \delta' \in GF(2)^m)$ is supposed to be approximately random, i.e. approximately $P(\Delta U = \delta | \chi(\Delta U) = \delta', \Delta X = \alpha) = p^{\mathrm{h}(\delta')}$, where $\delta = (\delta_1, \delta_2, \cdots, \delta_m), \chi(\delta) = \delta'$ because, for the input-differential of nonlinear layer $\Delta W = (\Delta W_1, \Delta W_2, \cdots, \Delta W_m) \in GF(2^n)^m$, each $P(\Delta W_p = \gamma_p | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p = 1/(2^n - 1)$ implies $P(\Delta U_p = \delta_p | \chi(\Delta U) = \delta', \Delta X = \alpha) = p = 1/(2^n - 1)$ and each $P(\Delta U_p = \delta_p | \Delta W_p = \gamma_p \neq 0, \Delta X = \alpha)$ $(\gamma = (\gamma_1, \gamma_2, \cdots, \gamma_m), \chi(\gamma) = \gamma')$ are jointly statistically independent from the definition of word-wise Markov cipher.

So we use this random output-differential cipher as an effective approximation model in the following discussion.

**Note.** Matsui assumed the randomness of input-differential of nonlinear layers $\Delta W = (\Delta W_1, \cdots, \Delta W_m)$, i.e.

$$P(\Delta W = \gamma | \chi(\Delta W) = \gamma', \Delta X = \alpha) = p^{\mathrm{h}(\gamma')} P(\chi(\Delta W) = \gamma' | \Delta X = \alpha)$$

instead of the randomness of output-differential of nonlinear layers $\Delta U = (\Delta U_1, \cdots, \Delta U_m)$, i.e.

$$P(\Delta U = \beta | \chi(\Delta U) = \beta', \Delta X = \alpha) = p^{\mathrm{h}(\beta')} P(\chi(\Delta U) = \beta' | \Delta X = \alpha)$$

in his presentation of [14]. The randomness of $\Delta W$ is a stronger assumption than the randomness of $\Delta U$, because, in the case of E2′-like transformation with 2-layer PSN-structures for example, the randomness of $\Delta W$ also implies the randomness of $\Delta U$. Furthermore, the randomness of $\Delta W$ may be too strong or even nonsense, because the randomness of input-differentials of linear layer $\Delta W = (\Delta W_1, \cdots, \Delta W_m)$ do not always yield the randomness of input-differentials of nonlinear layer $\Delta U = (\Delta U_1, \cdots, \Delta U_m)$ : Two input-differential words of nonlinear layers $\Delta W_{p_1}$, $\Delta W_{p_2}$ $(p_1 \neq p_2)$ may be both random, i.e.

$$P(\Delta W_{p_1} = \gamma_{p_1} | \Delta X = \alpha) = P(\Delta W_{p_2} = \gamma_{p_2} | \Delta X = \alpha) = p = 1/(2^n - 1)$$

but coincide, i.e. constantly $\Delta W_{p_1} = \Delta W_{p_2}$.

Therefore, we interpret Matsui's tacit assumption in his explanation as a random output-differential (Feistel) cipher.

### 3.4 Truncated Differential Probability

Using these definitions, we can accurately define the truncated differential probability. In this definition, as a cipher model, we consider a cipher with a random output-differential initial transformation $X(0) = g(X, Z^{(0)})$, and a random output-differential round function $X(i+1) = f(X(i), Z^{(i)})(i = 0, 1, \cdots, r-1)$ where $Z^{(i)}(i = 0, 1, \cdots)$ are sub-keys.

**Definition 8** *Let $X(0) = g(X, Z^{(0)})$ be an arbitrary random output-differential initial transformation and $X(i+1) = f(X(i), Z^{(i)})$ be a round function such that $X(r) = (f \circ \cdots \circ f \circ g)(X, Z^{(0)}, Z^{(1)}, \cdots, Z^{(r)})$ is also a random output-differential cipher for all $r$. An $i$-round truncated differential of $i$-round iterated cipher $X(r) = (f \circ \cdots \circ f)(X(0), Z^{(1)}, \cdots, Z^{(r)})$ is the couple $(\alpha', \beta')$, where $\alpha$ is the differential of a pair of distinct values $X(0)$ and $X^*(0)$, $\alpha' = \chi(\alpha)$ is the characteristic of $\alpha$; $\beta$ is a possible difference for the resulting $i$-th round outputs $X(i)$ and $X^*(i)$; $\beta' = \chi(\beta)$ is the characteristic of $\beta$. The probability of $i$-round truncated differential $(\alpha', \beta')$ is the conditional probability that $\beta'$ is the characteristic of difference $\Delta X(i)$ of the cipher text pair after $i$ rounds given that the characteristic of pair $(X(0), X(0)^*)$ has difference $\chi(\Delta X) = \alpha'$ when the plaintext $X$ and the sub-keys $Z^{(0)}, \cdots, Z^{(i)}$ are independent and uniformly random. We denote this truncated differential probability by $P'_i(\beta'(i), \beta'(0)) = P(\chi(\Delta X(i)) = \beta'(i) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha)$.*

This definition is well defined if we assume the random output-differential (Feistel) cipher. Without the assumption, this is not well-defined, because two input-differential values with same characteristic value do not always yield the same truncated differential probabilities. We assume this model as an effective approximation model of a word-wise Markov cipher.

## 4 Truncated Differential Probability and Differential Probability of Random Output-Differential (Feistel) Ciphers

### 4.1 Truncated Differential Probability of PSN-structures and Differential Probability of SPN-structures

In this subsection, we evaluate the truncated differential probability of the $2s$ layer PSN-structure and (the maximum average of) the differential probability of the $(2s+1)$ layer SPN-structure, where we assume all random functions are bijective. In this calculation, we first calculate the truncated differential probability of the $2s$ layer PSN-structure, and, using this probability, we calculate (the maximum average of) the differential probability of the $(2s+1)$ layer SPN-structure.

We assume the first nonlinear layer is a random output-differential (initial) transformation, and the round functions, which are composed of a linear layer and a nonlinear layer, i.e. 2-layer PSN-structures, is a random output-differential round function. We denote $\Delta X$ as the input-differential of the first nonlinear layer, $\Delta X(0)$ as the output-differential of the first nonlinear layer, $\Delta X(1)$ as the output-differential of the second nonlinear layer, $\cdots$, $\Delta X(s)$ as the output-differential of $(s+1)$-th nonlinear layer, $\Delta Y(0)$ as the input-differential of the first nonlinear layer, $\Delta Y(1)$ as the input-differential of the second nonlinear layer, $\cdots$, $\Delta Y(s)$ as the input-differential of the $(s+1)$-th nonlinear layer.

We denote the differential probability of $(2s+1)$-layer SPN-structures as

$$P_i(\beta(i), \alpha) = P(\Delta X(i) = \beta(i) | \Delta X = \alpha).$$

We denote the truncated differential probability of $2s$-layer PSN-structures as

$$P'_i(\beta'(i), \beta'(0)) = P(\chi(\Delta X(i)) = \beta'(i) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha).$$

The relation between differential probability and truncated differential probability can be represented as follows, where $\beta'(i) = \chi(\beta(i))$ for all $i = 0, 1, \cdots, s$,

$$\begin{aligned} &P_i(\beta(i), \alpha) = \\ &\quad \sum_{\beta'(0)} P(\Delta X(i) = \beta(i)) | \chi(\Delta X(i)) = \beta'(i), \Delta X = \alpha) \\ &\quad * \ \ P'_i(\beta'(i), \beta'(0)) * P(\chi(\Delta X(0) = \beta'(0)) | \Delta X = \alpha). \end{aligned}$$

In this case, if we assume the initial transformation is a random output-differential transformation and

$$P(\chi(\Delta X(0)) = \beta'(0) | \Delta X = \alpha) = \begin{cases} 1 & \text{if } \beta'(0) = \chi(\alpha) \\ 0 & \text{otherwise,} \end{cases}$$

as a natural approximation model of the first nonlinear layer, we can prove

$$P_i(\beta(i), \alpha) = p^{\mathrm{h}(\beta'(i))} P'_i(\beta'(i), \alpha'),$$

because

$$P(\Delta X(i) = \beta(i) | \chi(\Delta X(i)) = \beta'(i), \Delta X = \alpha) = p^{\mathrm{h}(\beta'(i))},$$

from the assumption of random output-differential cipher, where $p = 1/(2^n - 1)$ and h is the function that indicates the Hamming weight of the input value.

This relation clearly indicates the relationship between the differential probability and the truncated differential probability. From this relation we can easily calculate the differential probability from the truncated differential probability in the case of random output-differential cipher. This relation also implies that the possibility of truncated differential cryptanalysis is equivalent to the possibility of differential cryptanalysis, because the ratio of obtained probability to average probability do not change.

## 4.2 Procedure for Calculating Differential and Truncated Differential Probability of the SPN-structure

The procedure for calculating truncated differential probability and the maximum average of the differential probability in case of the SPN structure is as follows.

For this procedure, we define function $\mathrm{N}(P, \gamma, \delta)$ for $m \times m$ matrix $P$ over $GF(2^n)$ and $\gamma, \delta \in GF(2)^m$ by

$$\begin{aligned} \mathrm{N}(P, \gamma, \delta) = \#\{&(\Delta X, \Delta Y) \in (GF(2^n)^m)^2 \setminus \{0\} | \\ &\Delta Y = P\Delta X, \chi(\Delta X) = \gamma, \chi(\Delta Y) = \delta\}, \end{aligned}$$

For this calculation we define semi-order $\prec$ in $GF(2)^m$ as follows.

$$a \prec b \Leftrightarrow (\forall i; (a(i) = 1 \Rightarrow b(i) = 1)) \wedge (a \neq b)$$

where we denote $a(i)$ and $b(i)$ as the $i$-th significant bits of $a$ and $b$, respectively.

For $m \times m$ matrix $P$ over $GF(2^n)$ and $\gamma, \delta \in GF(2)^m$, we define

$$\begin{aligned}\mathrm{M}(P, \gamma, \delta) = \#\{(\Delta X, \Delta Y) \in (GF(2^n)^m)^2 \setminus \{0\} | \\ \Delta Y = P\Delta X, \chi(\Delta X) \preceq \gamma, \chi(\Delta Y) \preceq \delta\},\end{aligned}$$

and $\mathrm{N}(P, \gamma, \delta)$ can be calculated recursively, using the following relations.

$$\mathrm{N}(P, \gamma, \delta) = \mathrm{M}(P, \gamma, \delta) - \sum_{(\gamma', \delta') \prec (\gamma, \delta)} \mathrm{N}(P, \gamma', \delta')$$

In this case, we assume a random output-differential cipher. Under this assumption, we can prove the following lemma.

**Lemma 2**

$$\begin{aligned}&P'_i(\beta'(i), \beta'(0)) = \\ &\sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) p^{\mathrm{h}(\beta'(i-1))} P'_{i-1}(\beta'(i-1), \beta'(0)),\end{aligned}$$

*where* $p = 1/(2^n - 1)$.

**Proof.** From the assumption of a random output-differential cipher,

$$\begin{aligned}&P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ &= P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(i-1)) = \beta'(i-1), \Delta X = \alpha) \\ &* \quad P(\chi(\Delta X(i-1)) = \beta'(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ &= p^{\mathrm{h}(\beta'(i))} P(\chi(\Delta X(i-1)) = \beta'(i-1) | \chi(\Delta X(0)) = \beta'(0), \Delta X = \alpha) \\ &= p^{\mathrm{h}(\beta'(i-1))} P'_{i-1}(\beta'(i-1), \beta'(0)),\end{aligned}$$

where $\beta'(i-1) = \chi(\beta(i-1))$.

From the definition of N,

$$\begin{aligned}\mathrm{N}(P, \beta'(i), \beta'(i-1)) = \#\{(\Delta X(i), \Delta X(i-1)) \in (GF(2^n)^m \setminus \{0\})^2 | \\ \Delta X(i) = P\Delta X(i-1), \chi(\Delta X(i)) = \beta'(i), \chi(\Delta X(i-1)) = \beta'(i-1)\},\end{aligned}$$

.

Therefore,

$$\begin{aligned}&P'_i(\beta'(i), \beta'(0)) \\ &= \sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) P(\Delta X(i-1) = \beta(i-1) | \chi(\Delta X(0)) = \beta'(0)) \\ &= \sum_{\beta'(i-1)} \mathrm{N}(P, \beta'(i), \beta'(i-1)) p^{\mathrm{h}(\beta'(i-1))} P'_{i-1}(\beta'(i-1), \beta'(0)),\end{aligned}$$

This lemma, yields the following procedure.

1) **Computing the Number of All Differential Paths**
   For given $P$, calculate $N(P, \gamma, \delta)$ for every $\gamma, \delta \in GF(2)^m$.
   $\mathrm{M}(P, \gamma, \delta)$ can be easily calculated by simple rank calculation as follows.

$$
\begin{aligned}
&\mathrm{M}(P,\gamma,\delta)\\
&= \#\{(\Delta X,\Delta Y)\in GF(2^n)^{2m}\setminus\{0\} | P\Delta X=\Delta Y, F(\bar{\gamma})\Delta X=0, F(\bar{\delta})\Delta Y=0\}\\
&= 2^{n*\dim\{(\Delta X,\Delta Y)\in GF(2)^{2m}\setminus\{0\}|P\Delta X=\Delta Y,F(\bar{\gamma})\Delta X=0,F(\bar{\delta})\Delta Y=0\}}-1\\
&= 2^{n(2m-\mathrm{rank}\left(\begin{array}{cc} P & E\\ F(\bar{\gamma}) & O\\ O & F(\bar{\delta})\end{array}\right))}-1,
\end{aligned}
$$

where $\bar{\gamma}$ and $\bar{\delta}$ are the complements of $\gamma$ and $\delta$, respectively, $E$ is an identity matrix, and $F(\bar{\gamma})$, $F(\bar{\delta})$, denote the diagonal matrices over $GF(2^n)$ whose $(i,i)$ component equals the $i$-th significant bit of $\bar{\gamma}$, $\bar{\delta}$ for $i=1,\cdots,m$, respectiveley.

$\mathrm{N}(P,\gamma,\delta)$ can be calculated recursively from the values of $\mathrm{M}(P,\gamma,\delta)$, using the following relation.

$$
\mathrm{N}(P,\gamma,\delta)=\mathrm{M}(P,\gamma,\delta)-\sum_{(\gamma',\delta')\prec(\gamma,\delta)}\mathrm{N}(P,\gamma',\delta')
$$

2) **Initialization**
For given $\alpha'\in GF(2)^m$, calculate $P'_0(\beta'(0),\alpha')$ for every $\beta'(0)\in GF(2)^m$, where

$$
P'_0(\beta'(0),\alpha')=\begin{cases}1 & \text{if } \beta'(0)=\alpha'\\ 0 & \text{otherwise,}\end{cases}
$$

3) **Recursive Computation of Truncated Differential Probability**
Utilizing the values of $N(P,\gamma,\delta)$, calculate $P'_i(\beta'(i),\alpha')$ recursively for every $\beta'(i)\in GF(2)^m$.

$$
\begin{aligned}
&P'_i(\beta'(i),\alpha')=\\
&\sum_{\beta'(i-1)}\mathrm{N}(P,\beta'(i),\beta'(i-1))p^{\mathrm{h}(\beta'(i-1))}P'_{i-1}(\beta'(i-1),\alpha')
\end{aligned}
$$

4) **Calculation of (Maximum Average of) Differential Probability**
Evaluate $P_i(\beta(i),\alpha)$ by

$$
P_i(\beta(i),\alpha)=p^{\mathrm{h}(\beta'(i))}P'_i(\beta'(i),\alpha')
$$

With this procedure we can compute the truncated differential probability of PSN-structures and (the maximum average of) the differential probability of SPN-structures with 16 input words. Furthermore, applying this procedure to the each MixColumn transformations of Rijndael, allows us to compute the truncated differential probability and (the maximum average of) the differential probability. From this computation, the maximum average of the differential probability of 7-layer Rijndael including 4 nonlinear layers, i.e. 4-round Rijndael, is upper-bounded by $1.00*p^{16}$ $(=1.065*2^{-128})$ and that of 9-layer Rijndael including 5 nonlinear layers, i.e. 5-round Rijndael, is upper-bounded by $0.940*p^{16}$ $(=1.0007*2^{-128})$ [1]. To be secure against differential and truncated differential cryptanalysis, 2 more layers (1 round) are necessary to avoid the exhaustive search of the the last 2 layers (1 round). This implies a total of 80 S-boxes is needed.

[1] [12] stated that 5-round differential with probability $1.06*2^{-128}$ was found, but this was typo. The correct round is 4.

## 4.3 Truncated Differential Probability of E2′-like Transformation

Using the values of the differential probability of the $2r$-layer PSN-structures, we can calculate the truncated differential probability of E2′-like transformations recursively. In this calculation, we assume the random output-differential Feistel cipher, hence the probabilities for $\chi(\Delta x \oplus \Delta y) = 1$ and $\chi(\Delta x \oplus \Delta y) = 0$ for two random output-differential values $\Delta x, \Delta y, \in GF(2^n) \setminus \{0\}$ are $(2^n - 2)/(2^n - 1)$ and $1/(2^n - 1)$, respectively.

The procedure for calculating the truncated differential probability of the E2′-like transformation is as follows.

1) **Computation of Truncated Differential Probability of Round Functions**
Using the procedure for calculating truncated differential probability of $2r$-layer PSN-structure, calculate the truncated differential of round functions. Hereafter, we denote the truncated differential probability of the $i$-th round function for the truncated differential $(\zeta'(i), \zeta'(i-1))$ by $Q'_r(\zeta'(i), \zeta'(i-1)) = P(\chi(\Delta X(i)) = \zeta'(i) | \chi(\Delta X(i-1)) = \zeta'(i-1))$ for $\zeta'(i), \zeta'(i-1) \in GF(2)^m$

2) **Initialization**
Let $\zeta'(0) = (\Delta L'(0), \Delta R'(0)) \in GF(2)^{2m}$. For given $\eta' \in GF(2)^{2m}$, calculate $P'_0(\zeta'(0), \eta')$ for every $\zeta'(0) \in GF(2)^{2m}$, where we assume

$$P'_0(\zeta'(0), \eta') = \begin{cases} 1 & \text{if } \zeta'(0) = \eta' \\ 0 & \text{otherwise,} \end{cases}$$

3) **Recursive Computation of Truncated Differential Probability**
Let $\zeta'(i) = (\Delta L'(i), \Delta R'(i)) \in GF(2)^{2m}$, $\zeta(i) = (\Delta L(i), \Delta R(i)) \in GF(2)^{2m}$, where $\chi(\zeta(i)) = \zeta'(i), \chi(\Delta L(i)) = \Delta L'(i), \chi(\Delta R(i)) = \Delta R'(i)$. Utilizing the values of truncated differential probabilities of round functions, calculate $P'_i(\zeta'(i), \eta')$ recursively for every $\zeta'(i) \in GF(2)^{2m}$.

$$P'_i(\zeta'(i), \eta') = \sum_{\substack{\xi', \\ \chi(L(i-1) \oplus \xi) = \Delta R'(i), \\ \chi(\xi) = \xi'}} Q'_i(\xi', \Delta R'(i-1)) P'_{i-1}(\zeta'(i-1), \eta')$$

4) **Calculation of (Maximum Average of) Differential Probability**
Calculate $P_i(\zeta(i), \eta)$ by

$$P_i(\zeta(i), \eta) = p^{\mathrm{h}(\zeta'(i))} P'_i(\zeta'(i), \eta'),$$

where $\chi(\eta) = \eta'$.

## 4.4 (Maximum Average of) Differential and Truncated Differential Probability of E2′-like Transformation

In this subsection, we evaluate the maximum average of the differential probability of E2′-like transformations with proper initial transformations, where we assume the all linear layers are same as that of E2.

First we consider E2′-like transformations with 2-layer PSN-structures. In this case, a nonlinear layer with 16 nonlinear functions, or 2-round E2′-like transformations with 2-layer PSN-structures can be adopted as the approximately random output-differential initial transformation. 8-round E2′-like transformation with 2-layer PSN-structures with proper initial transformation has maximum average of differential probability of less than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 80 nonlinear functions. To offer security against differential cryptanalysis, 2 more rounds are necessary, which means it needs a total of 96 nonlinear functions.

If we slightly change linear transformation of SPN-structures, it can be provably secure with 72 nonlinear functions. To offer security against differential cryptanalysis, 2 more rounds are necessary, which means it needs a total of 88 nonlinear functions.

Next we consider E2′-like transformations with 4-layer PSN-structures. In this case, a nonlinear layer with 16 nonlinear functions or 2-round E2′-like transformations with 2-layer or 4-layer PSN-structures can be adopted as the proper initial transformation. A 5-round E2′-like transformation with 4-layer PSN-structures with proper initial transformation has maximum average of differential probability lower than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 96 nonlinear functions. To be secure against differential cryptanalysis, 1 more round is necessary, which means it needs a total of 112 nonlinear functions to avoid the exhaustive search of the final round.

On the other hand, an 8-round E2-like transformation with 3-layer SPN-structures, has maximum average of differential probability lower than $0.940 * p^{16}$ $(= 1.0007 * 2^{-128})$. In this case, it is provably secure with 128 S-boxes (in this case, approximately random output-differential initial function is not necessary because of the first nonlinear layers of the first and second S-boxes). To be secure against differential cryptanalysis, 1 more round is necessary, considering the exhaustive search of the final round, which implies it needs a total of 144 S-boxes.

These results means that E2′-like transformations with 2-layer PSN-structures is more secure than 3 or 4 layer.

The block cipher MISTY with 16-input words and 3-rounds has maximum average of differential probability equal to $p_{\max}^{16}$, where $p_{\max}$ is the maximum average of differential probability of nonlinear functions. In this case, it is provably secure with 81 S-boxes. To be secure against differential cryptanalysis, 1 more round is necessary, which implies it needs a total of 108 S-boxes.

### 4.5 Impossible Differential Cryptanalysis of Rijndael, E2′-like Transformation

Impossible differential cryptanalysis is a cryptanalysis against block ciphers which utilizes the pair of input and output-differentials whose differential probability equals 0 [1].

In the previous procedure, we proposed the procedure which calculates the truncated differential probability of random output-differential (Feistel) ciphers. It follows that from the relations between truncated differential probability and differential probability we can also calculate the differential probability.

From the values of the differential probability, our procedure can calculate the resistance against impossible differential cryptanalysis, by counting the number of differentials whose probabilities equal 0. In the case of E2′-like transformations with 2-layer PSN-structures, it can be proved that 9-rounds offer security against impossible differential cryptanalysis while 8-rounds do not. In the case of E2-like transformations with 3-layer SPN-structures, 8-rounds offer security against impossible differential cryptanalysis and 7-rounds do not. Comparing the numbers of nonlinear functions, E2′-like transformations with 2-layer PSN-structures is superior to E2-like transformations with 3-layer SPN-structures, i.e. the basic structure of block cipher E2.

In the case of Rijndael, it can be proved that 7-layers (including 4 nonlinear layers) offers security against impossible differential cryptanalysis while 5-layers (including 3 nonlinear layers) do not. Comparing the numbers of nonlinear functions, basic structure of Rijndael has a little higher level of security against impossible cryptanalysis than E2′-like transformation with 2-layer PSN-structures. However, considering the amount of linear layer operations, E2′-like transformations with 2-layer PSN-structures may be superior to the basic structure of Rijndael, because the linear layer of E2′-like transformations consists of only "xor" whereas that of Rijndael consists of heavier linear transformation over Galois field $\mathrm{GF}(2^8)$.

# 5 Conclusion

This paper examined the truncated differential probability and the differential probability of the word-oriented Markov ciphers and random output-differential (Feistel) ciphers like Rijndael and (modified) E2 and clarified the relations among the differential, truncated differential and the impossible differential cryptanalysis of the random output-differential (Feistel) cipher. This random output-differential (Feistel) cipher uses a weaker assumption than the assumption that all S-box differentials are equally likely. This is not a strong assumption because denying this model is equivalent to denying the truncated differential cryptanalysis. We then described a procedure for calculating the truncated differential probability and (maximum average of) the differential probability of such ciphers. Using this procedure, we computed and proved the security of Rijndael, E2 and the E2′-like transformation against differential, truncated differential and impossible differential cryptanalyses under the assumption of a random output-differential (Feistel) cipher. Our evaluation finds that Rijndael is the most secure, and the E2′-like transformation with 2-layer PSN structure is a little less secure. However, the linear transformation in E2′-like transformations is lighter than that of Rijndael and can be improved by slightly changing, so the overall speed may be the highest (may be "not" the highest). Our results implies that SPN-structures (like Rijndael, Serpent) and Feistel structures with S-boxes composed of 2-layer PSN-structures (like E2-like transformation with 2-layer PSN-structures) have no disadvantage in terms of security against differential and truncated differential cryptanalysis. We can similary evaluate the security of Feistel structures with S-boxes composed of 2-layer SPN-structures (like Twofish [18]) against differential and truncated differential cryptanalysis, though we have not evaluated Twofish yet because Twofish is not composed of just word-wise operations of fixed size. However, Feistel structures with 2-layer SPN-structures can be proved to be secure and have no disadvantage in terms of security against differential and truncated differential cryptanalysis, if we select the proper linear transformations in their SPN-structures.

# References


[1] E.Biham, A.Biryukov and A.Shamir, "Cryptanalysis of Skipjack Reduced to 32 Rounds Using Impossible Differentials." J. Stern (Ed.): EUROCRYPT'99, LNCS 1592, pp. 12-23, Springer-Verlag, Berlin, 1999.

[2] E.Biham and A.Shamir. "Differential Cryptanalysis of DES-like Cryptosystems." Journal of Cryptology, Vol.4, No.1, pp.3-72, 1991. (The extended abstract was presented at CRYPTO'90).

[3] J. Daemen and V. Rijmen. "AES Proposal Rijndael," AES Round 1 Technical Evaluation CD-1: Documentation, National Institute of Standards and Technology, Aug 1998. See **http://www.nist.gov/aes**.

[4] T.Jakobsen and L.R.Knudsen. "The Interpolation Attack on Block Cipher." In E.Biham, editor, Fast Software Encryption - 4th International Workshop, FSE'97, Volume 1267 of Lecture Notes in Computer Science, pp.28-40, Berlin, Heidelberg, NewYork, Springer-Verlag, 1997.

[5] L.R. Knudsen and T.A. Berson, "Truncated Differentials of SAFER." In Fast Software Encryption - Third International Workshop, FSE'96, Volume 1039 of Lecture Notes in Computer Science, Berlin, Heidelberg, NewYork, Springer-Verlag, 1996.

[6] M. Kanda et al. "A New 128-bit Block Cipher E2" Technical Report of IEICE. ISEC98-12.

[7] X. Lai, J. L. Massey and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," Advances in Cryptography-EUROCRYPTO '91. Lecture Notes in Computer Science, Vol. 576. Springer-Verlag, Berlin, 1992, pages. 86-100.

[8] M.Matusi, "Linear Cryptanalysis Method for DES Cipher." In T. Helleseth, editor, Advances in Cryptology - EUROCRYPT'93, Volume765 of Lecture Notes in Computer

Science, pp.386-397. Springer-Verlag, Berlin, Heidelberg, NewYork, 1994. (A preliminary version written in Japanese was presented at SCIS93-3C).

[9] M. Matsui, "New structure of block ciphers with provable security against differential and linear cryptanalysis," In Dieter Grollman, editor, Fast Software Encryption: Third International Workshop, volume 1039 of Lecture Notes in Computer Science, pages 205-218, Cambridge, UK, 21-23 February 1996. Springer-Verlag.

[10] M. Matsui, "New block encryption algorithm MISTY." In Eli Biham, editor, Fast Software Encryption: 4th International Workshop, volume 1267 of Lecture Notes in Computer Science, pages 54-68, Haifa, Israel, 20-22 January 1997. Springer-Verlag

[11] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, "Handbook of Applied Cryptography", CRC Press, 250-250 (1997).

[12] Moriai, S., Sugita, M., Aoki, K., Kanda, M. "Security of E2 against truncated Differential Cryptanalysis" Sixth Annual Workshop on Selected Areas in Cryptography (SAC'99), pages, 133-143 (1999).

[13] Moriai, S. et al. "Security of E2 against truncated Differential Cryptanalysis" Technical Report of IEICE, to appear.

[14] Matsui, M. and Tokita, T. "Cryptanalysis of a Reduced Version of the Block Cipher E2" in 6-th international workshop, preproceedings FSE'99

[15] K. Nyberg and L. R. Knudsen, "Provable security against a differential attack," in Advances in Cryptology - EUROCRYTO'93, LNCS 765, pages 55-64, Springer-Verlag, Berlin, 1994.

[16] M. Sugita, "Security of Block Ciphers with SPN-Structures" Technical Report of IEICE. ISEC98-30.

[17] M. Sugita, K. Kobara, H. Imai, "Pseudorandomness and Maximum Average of Differential Probability of Block Ciphers with SPN-Structures like E2." Second AES Workshop, 1999.

[18] B. Schneier, J. Kelsey, D. Whiting, D. Wagner and C. Hall, "Twofish: A 128-Bit Block Cipher," AES Round 1 Technical Evaluation CD-1: Documentation, National Institute of Standards and Technology, June 1998. See **http://www.nist.gov/aes**.

[19] T. Tokita, M. Matsui, "On cryptanalysis of a byte-oriented cipher", The 1999 Symposium on Cryptography and Information Security, pages 93-98 (In Japanese), Kobe, Japan, January 1999.